Portal User Interface Framework Guide

Portal User Interface Framework Guide

Overview

This document details how the portal framework turns a portal you develop in WebLogic Workshop (see Figure 1) into the portal desktop visitors see in a browser (see Figure 2). The goal of describing the portal framework is to help you develop and troubleshoot your portals. These topics enable you to look at a rendered portal in a browser and understand which pieces of the underlying framework you need to modify to get the results you want. In addition, the Look & Feel Editor is discussed. The Look & Feel Editor lets you interactively modify the text styles used by a portal.

The topics in this document describe key portal framework components and walk you through the portal rendering process. These topics include:

- How Look & Feel Determines Rendering

  Describes how look & feel determines how a portal desktop is rendered and what it looks like.

- How the Shell Determines Header and Footer Content

  Describes how shells determine the content of a desktop header and footer.

- How Portal Components Are Rendered

  Illustrates the rendering process, showing how a portal component is converted to HTML.

- The Look & Feel Editor

  Discusses functional parts of the Look & Feel Editor and important concepts that will help you use the Editor effectively, such as CSS inheritance.

Figure 1 Portal file under development in WebLogic Workshop.

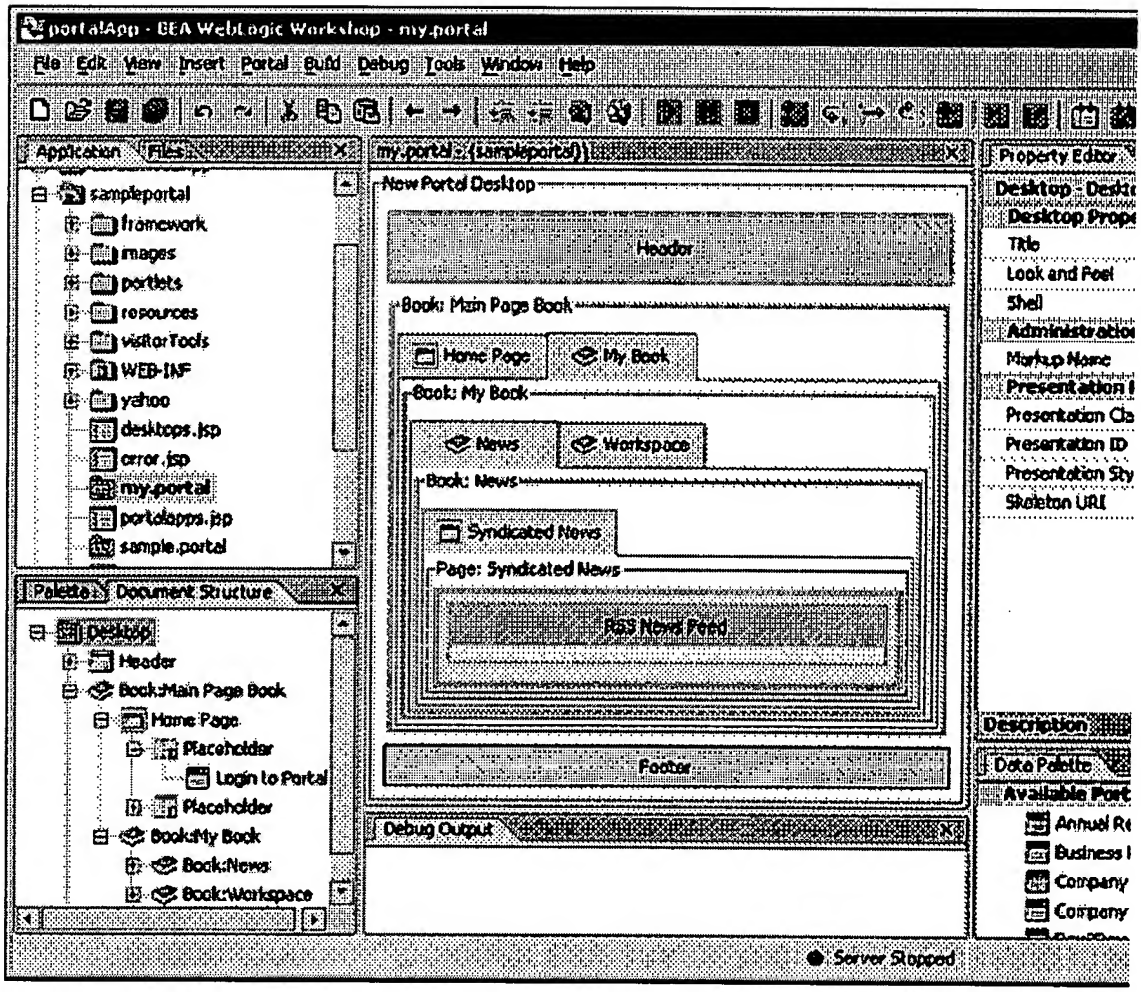
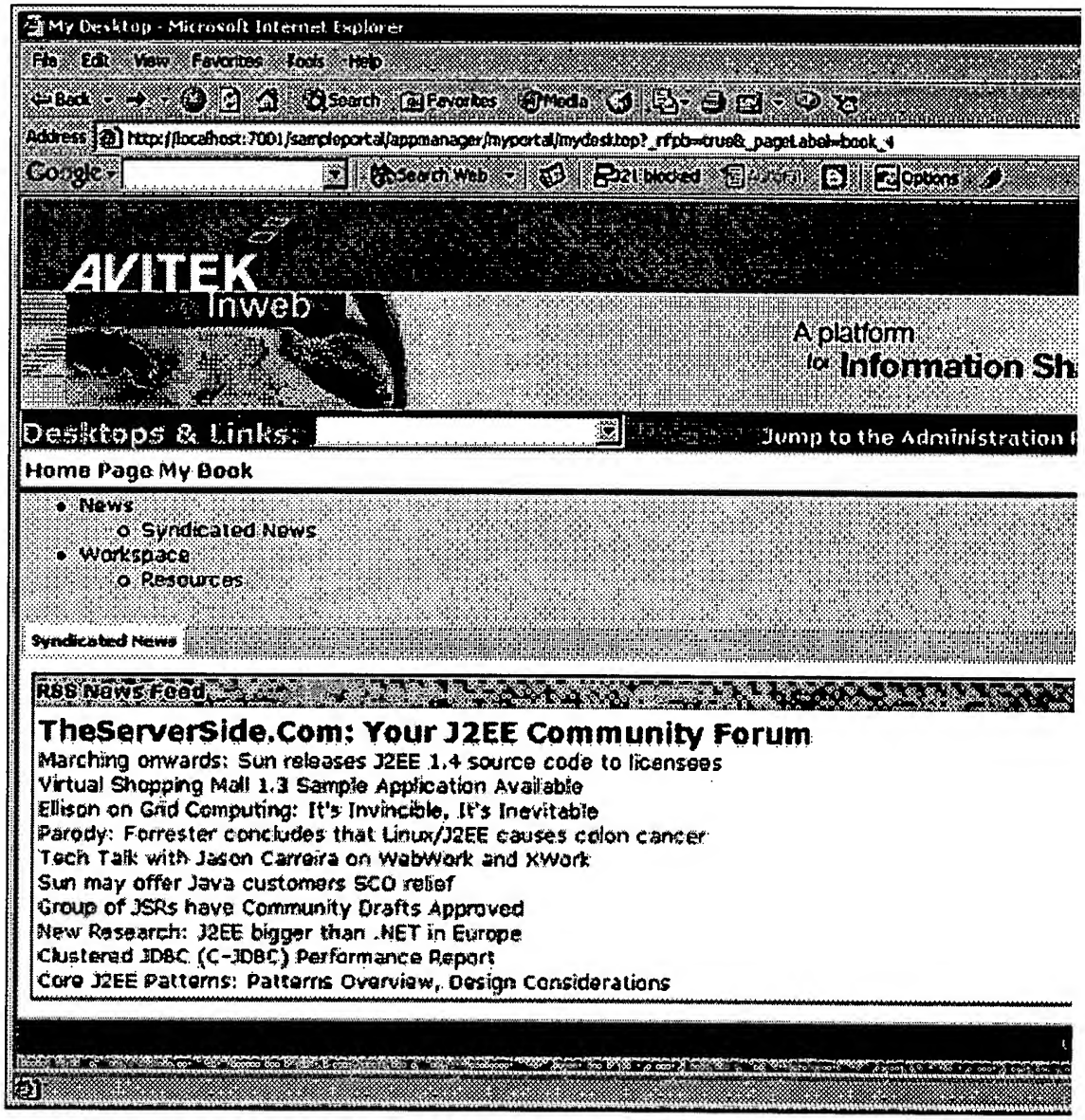


Figure 2 Portal rendered in HTML.



How Look & Feel Determines Rendering

When you build a portal in WebLogic Workshop, the look & feels you use are the key to how your portal is rendered and what it looks like when it is rendered. This topic shows you how the different pieces of the look & feel framework are combined and configured to provide what the portal framework needs to render the look & feel in HTML.

This topic contains the following sections:

- Overview

- The Look & Feel File
- The Portal File
- Location of the Look & Feel Resources
- The `skin.properties` File
- Look & Feel Overrides
- Summary

Overview

The look & feel encompasses the following:

- **Skin** - A skin is a group of Cascading Style Sheet (CSS) files, framework images (mainly for portlet title bar icons), and JavaScript functionality that is used in the portal desktop when it is rendered in HTML. A portal Web project can have multiple skins. When you select a look & feel for a desktop, a specific skin is used. Following are example skin elements, Image, CSS Style, and JavaScript Functions:

Image


CSS Style

.bea-portal-button-float
{
}
.bea-portal-button-float img
{
vertical-align: top;
margin: 1px;
border-style: solid;
border-width: 1px;
border-color: #666699;
}

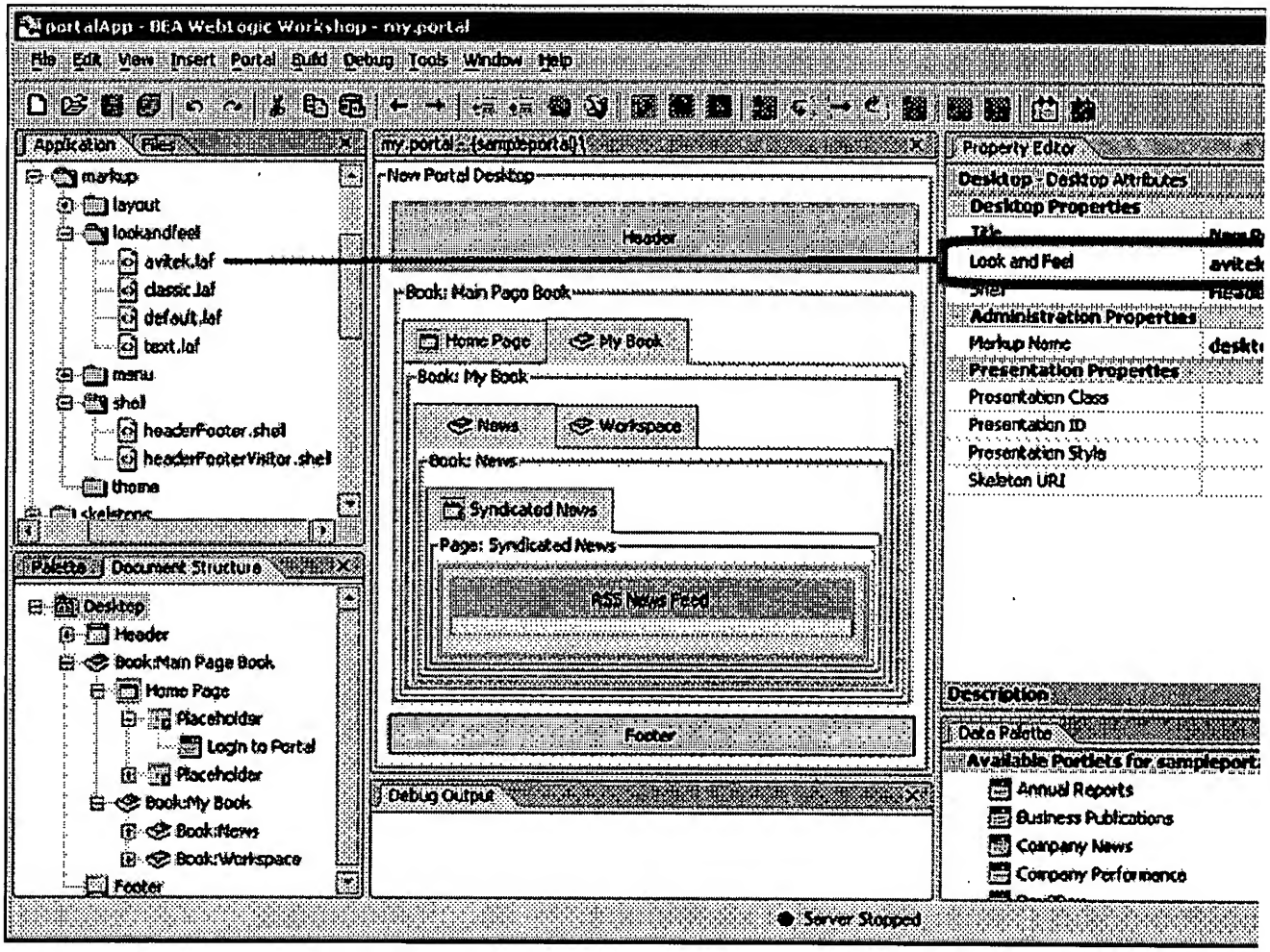
JavaScript Function

```
function initPortletFloatButtons()

{

  var links =
    document.getElementsByTagName("a");
    for (var i = 0; i < links.length; i++)
       {
         if (links[i].className &&
             links[i].className ==                "bea-portal-button-fl
              {

          initPortletFloatButton(links[i]);
              }
       }

}
```

- **Skeleton JSPs** - A skeleton is a group of JSPs that are used to render each component of the portal desktop as HTML, from the desktop to books and pages to portlet title bars. The skeleton provides the physical boundaries of the portal components and provides references to the images, CSS classes, and JavaScript functions from the skin needed to render the portal. A portal Web project can have multiple skeletons. When you select a look & feel for a desktop, a specific skin and skeleton is used.

A look & feel is represented by an XML file (with a `.laf` extension). As shown in the following figure, the `.laf` (`avitek.laf`) file is located in the `lookandfeel` folder of a portal project. In addition, the `.laf` file name (for example, `avitek`) can be selected in the Desktop properties panel.

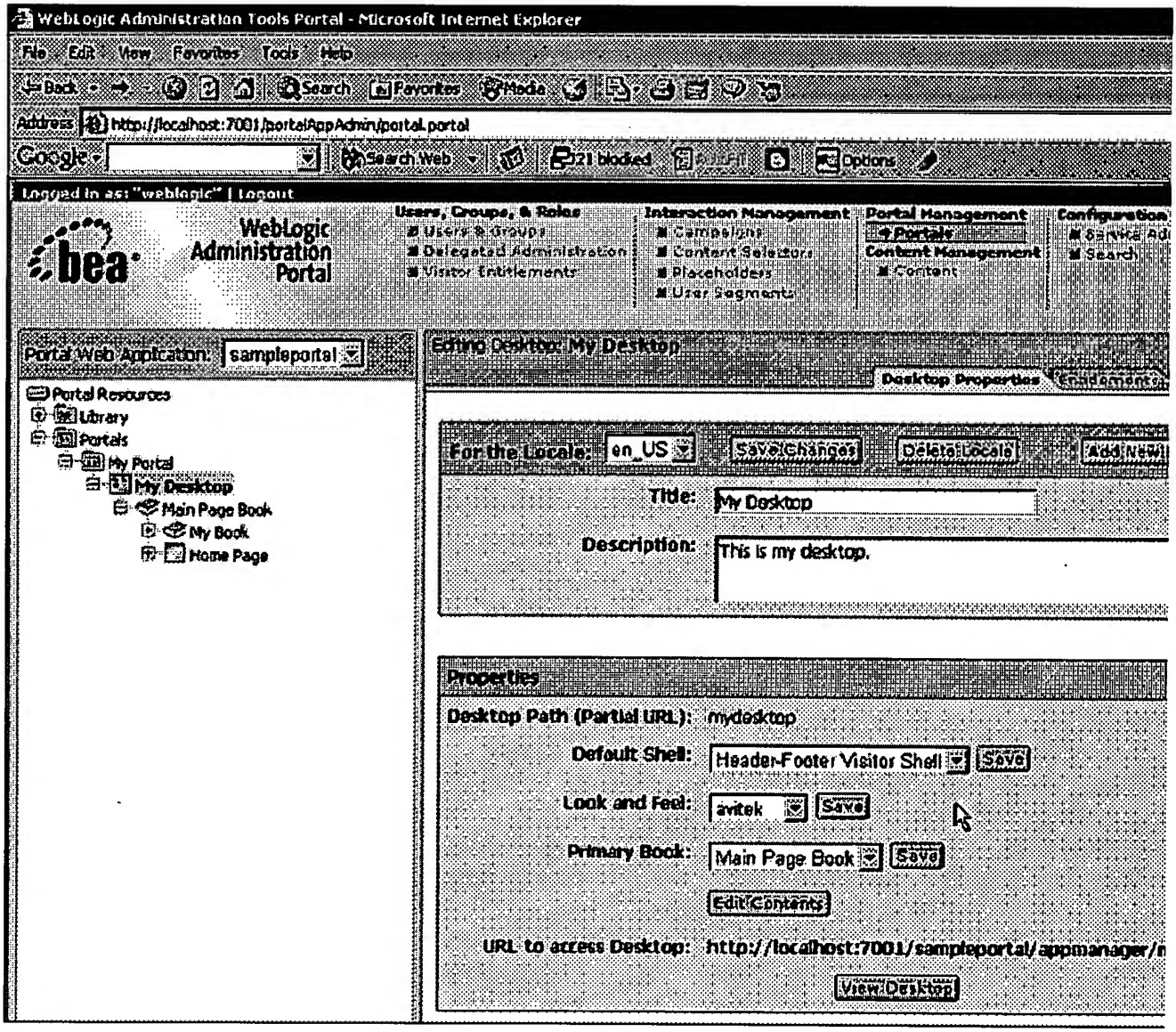
Figure 3 Portal rendered in HTML.



Developers building portals with WebLogic Workshop are not the only users who can select the look & feel used by a portal desktop. While developers create look & feels and select the default look & feel used by a portal, portal administrators and visitors may ultimately determine the desktop look & feel. The following figures show how portal administrators and users can change the look & feel used by the desktop.

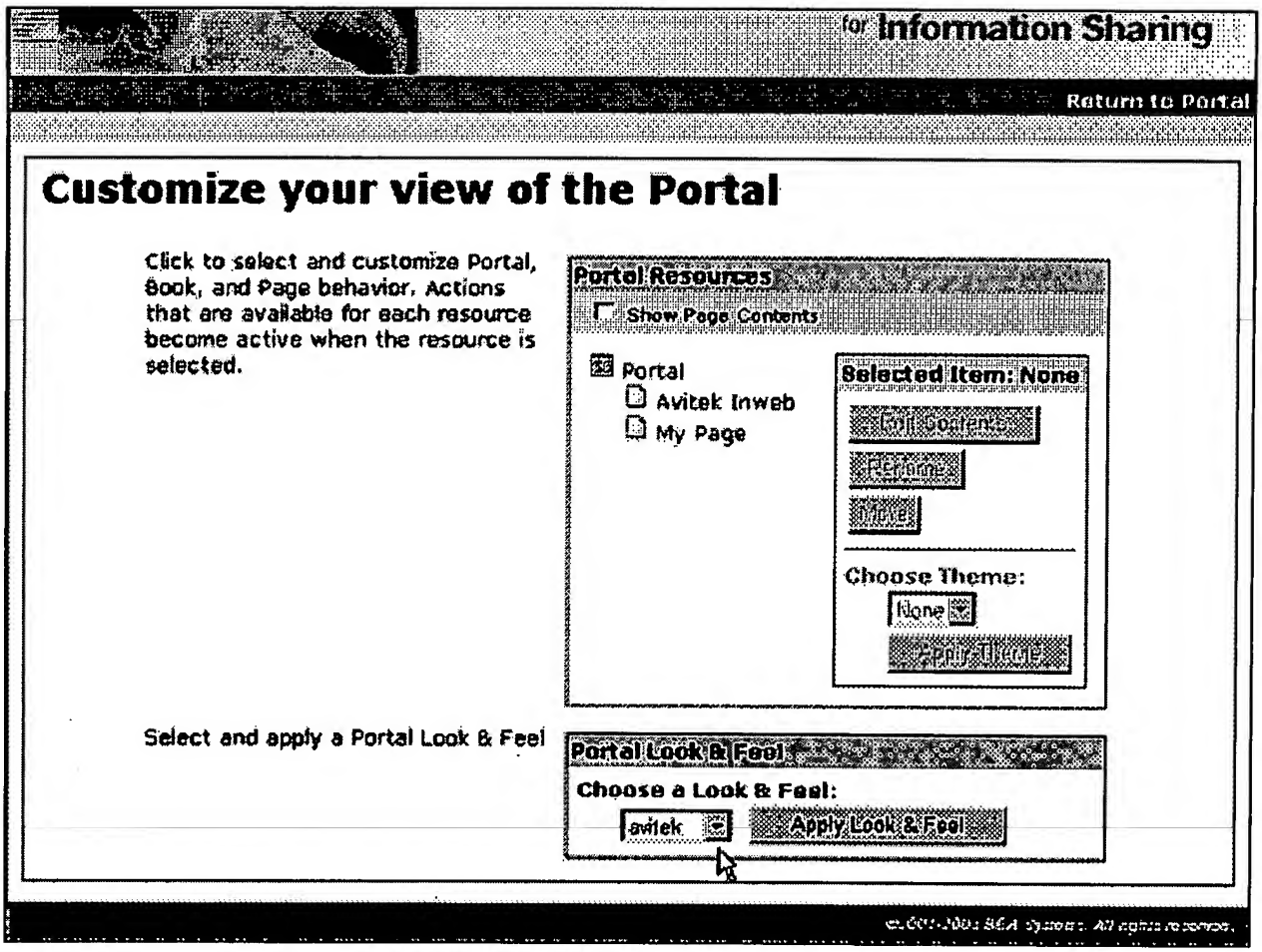
After a portal administrator creates a desktop in the WebLogic Administration Portal, the administrator can change the desktop look & feel on the Desktop Properties page, as shown in the following figure.

Figure 4 The Desktop Properties page



If visitor tools are enabled for the desktop, visitors can click the "Customize My Portal" link and change the desktop look & feel, as shown in the following figure.

Figure 5 Customizing a portal



The following section shows the contents of a look & feel XML-based `.laf` file and describes how it is used as the basis of portal desktop rendering.

The Look & Feel File

Look & feel files point to the specific skin and skeleton to be used for the overall desktop look & feel.

Look & feel files are stored in the following location: `<portal_Web_project>/framework/markup/lookandfeel`. Following is the `avitek.laf` provided by BEA. The key attributes are highlighted.

```
<?xml version="1.0" encoding="UTF-8"?>

<netuix:markupDefinition xmlns:netuix="http://www.bea.com/servers/netuix/xsd/con

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.bea.com/servers/netuix/xsd/controls/netuix/1.0.0

    <netuix:locale language="en"/>
```

```
    <netuix:markup>
        <netuix:lookAndFeel
            definitionLabel="avitek" title="avitek"
            description="The avitek look and feel"
            skin="avitek" skinPath="/framework/skins/"
            skeleton="default" skeletonPath="/framework/skeletons/"
            markupType="LookAndFeel" markupName="avitek"/>
    </netuix:markup>
</netuix:markupDefinition>
```

The following table describes the look & feel file's key attributes.

Table 1 Look & Feel File Attributes

Attribute	Description
definitionLabel	Required. The unique label used to identify the look & feel for setting entitlements. Each look & feel in the portal Web project must have a unique `definitionLabel`. For best practices, use the same name as the `markupName`.
title	Required. The string used to display the name in the Look & Feel drop-down fields in WebLogic Workshop, the WebLogic Administration Portal, and on the visitor tools page.
description	Optional. Description of the look & feel. The description is used in the WebLogic Administration Portal; when you select a look & feel in the portal Library, the description appears on the Look & Feel Properties page.
skin	Optional. The name of the directory containing the skin you want to use. If you do not set this attribute, the `/framework/skins/default` skin is used.
skinPath	Optional. The path, relative to the portal Web project, to the parent directory of the skin directory. If you do not set this attribute, the `/framework/skins/<skin_attribute_name>` skin is used. If no skin attribute is set, the `/framework/skins/default` skin is used.
skeleton	Optional. The name of the directory containing the skeleton JSPs you want to use. If you do not set this attribute, the framework uses the `default.skeleton.id` path in the `skin.properties` file of the skin used. If you do not set this attribute and no `default.skeleton.id` path is set in `skin.properties`, the `/framework/skeletons/default` skeleton is used.

skeletonPath	Optional. The path, relative to the portal Web project, to the parent directory of the skeleton directory. If you do not set this attribute, the framework uses the `default.skeleton.path` in the `skin.properties` file of the skin is used. If you do not set this attribute and no `default.skeleton.path` is set in `skin.properties`, the `/framework/skeletons/<skeleton_attribute_name>` skeleton is used. If you do not set this attribute and no skeleton attribute is set, the `/framework/skeletons/<default.skeleton.id>` skeleton is used. If you do not set this attribute and no skeleton attribute is set, and `skin.properties` contains no `default.skeleton.id`, the `/framework/skeletons/default` skeleton is used.
markupType	Required. The name of the type of component. Must always be "LookAndFeel".
markupName	Required. The name for the look & feel. Each look & feel in the portal Web project must have a unique `markupName`. For best practices, use the same name as the `definitionLabel`.

When you select a Look & Feel in the WebLogic Workshop Property Editor for a selected desktop, the look & feel XML is automatically added to the underlying XML in the `.portal` file, as shown in the following section.

The Portal File

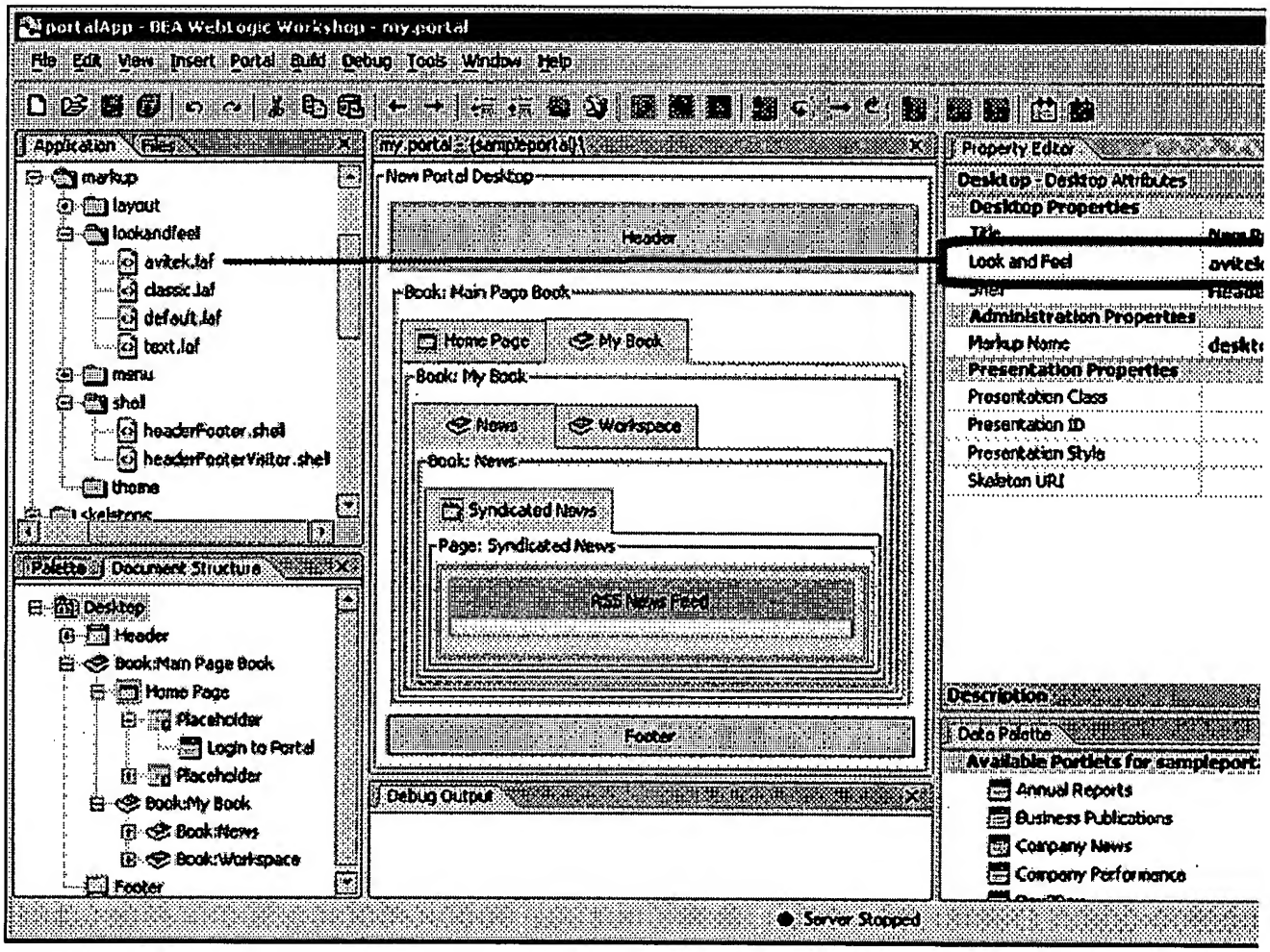
The following example portal file, created with the Portal Designer, shows the inserted look & feel XML (in bold) from the `.laf` file. The portal file is a template with which multiple desktops can be created in the WebLogic Administration Portal. When used as a template, the portal file determines the default look & feel of any desktop created from it.

```
<?xml version="1.0" encoding="UTF-8"?>

<portal:root xmlns:html="http://www.w3.org/1999/xhtml-netuix-modified/1.0.0"

    xmlns:netuix="http://www.bea.com/servers/netuix/xsd/controls/netuix/1.0.0"

    xmlns:portal="http://www.bea.com/servers/netuix/xsd/portal/support/1.0.0"

    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

    xsi:schemaLocation="http://www.bea.com/servers/netuix/xsd/portal/support/1.0
    portal-support-1_0_0.xsd">

    <portal:directive.page contentType="text/html;charset=UTF-8"/>

    <netuix:desktop definitionLabel="defaultDesktopLabel" markupName="desktop"
```

```
markupType="Desktop" title="New Portal Desktop">

  <netuix:lookAndFeel definitionLabel="avitek" description="The avitek loo
    and feel"

      markupName="avitek" markupType="LookAndFeel" skeleton="default"

      skeletonPath="/framework/skeletons/" skin="avitek"
         skinPath="/framework/skins/" title="avitek"/>

<netuix:shell description="A header with a link and footer is included in

      markupName="headerFooterVisitor" markupType="Shell"
        title="Header-Footer Visitor Shell">

      <netuix:head/>

      <netuix:body>

          <netuix:header>

              <netuix:jspContent contentUri="/portlets/header/header.jsp"/

          </netuix:header>


              [XML for books, pages, and portlets...]

          <netuix:footer>

              <netuix:jspContent contentUri="/portlets/footer/footer.jsp"/

          </netuix:footer>
```

The look & feel XML is inserted when the Look & Feel property is set for the selected desktop in the Property Editor. For example, in the following figure, the look & feel called **avitek** is selected.

Figure 6 Selecting a Look & Feel

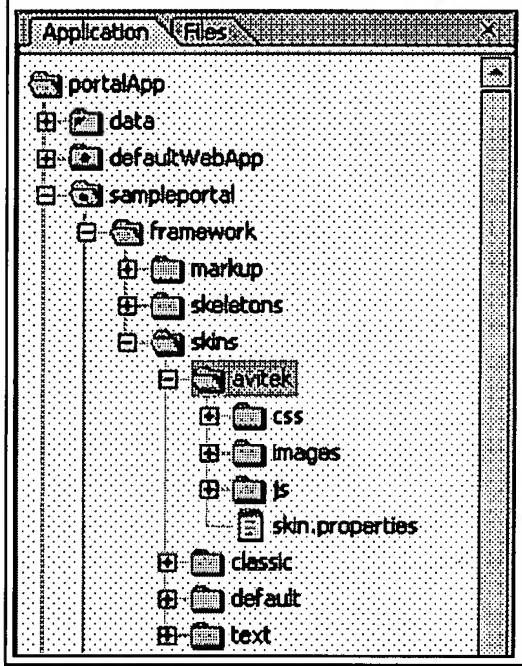


When the `.laf` file is inserted into the `.portal` file, its job is finished in the rendering process and the `.portal` file is used to set look & feel.

Location of the Look & Feel Resources

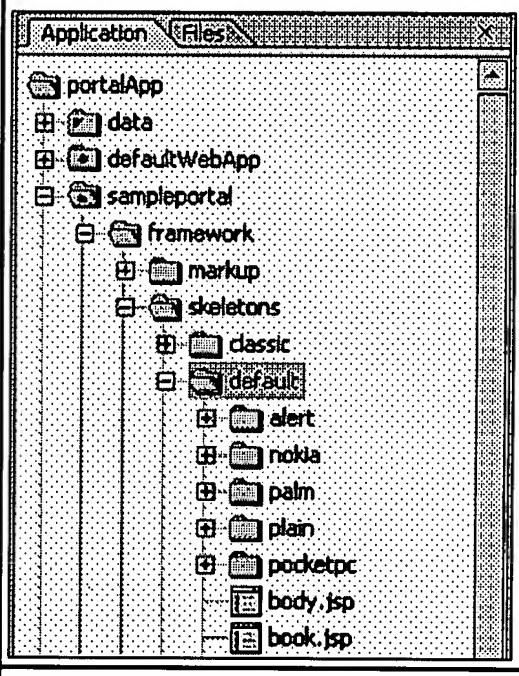
The look & feel attributes in the portal file tell the portal which skin and skeleton to use to render the portal in HTML. The portal in the previous example will use the following skin and skeleton resources:

Skin	Skeleton



The `/css`, `/images`, and `/js` directories contain the CSS files, framework images (mainly portlet titlebar icons), and JavaScript files that will be used in the skin. The `skin.properties` file (discussed in the next section) contains references to these resources, and at rendering time those resource references are inserted into the HTML `<head>` region. You can name your skin resource directories anything you like as long as you reference them correctly in `skin.properties` (or `skin_custom.properties`).

Skins can also contain subdirectories for sub-skins, or themes (discussed in Look & Feel Overrides).



The skeleton is made up of JSPs that map to and convert each portal component to HTML. The XML elements for the portal components in the `.portal` file determine the order in which the skeleton JSPs are called and rendered as HTML.

This figure shows a clipped view of the skeleton contents. The subdirectories shown are skeleton themes and skeletons used for mobile devices. The JSPs in the `/default` directory make up the "default" skeleton.

Themes are discussed in Look & Feel Overrides.

Note About Portlet Titlebar Icons

The icons used in portlet title bars are stored in the skin's `/images` directory. The portal framework reads the portal Web project's `WEB-INF/netuix-config.xml` file to determine which of these graphics to use for the portlet's different states and modes (minimize, maximize, help, edit, and so on).

The skin.properties File

Each skin has a `skin.properties` file, which is used by the portal framework to populate the `<head>` section of the rendered HTML, among other things. Included in `skin.properties` are references to the images directory, the CSS files containing the styles to be used in the

HTML, and the JavaScript files containing the functions that will be used in the HTML.

Note: You can also create a file called `skin_custom.properties` in the same directory as `skin.properties`. Any entries you include in `skin_custom.properties` are also added to the HTML `<head>` region. This feature lets you customize the properties without having them overwritten by product updates.

The following will not be rendered:

- Style Sheet styles that do not exist in one of the `.css` files listed in the `<head>`.
- JavaScript functions that do not exist in one of the `.js` files listed in the `<head>`.

That is why it is important to add references to all skin resources in `skin.properties` or `skin_custom.properties`.

The `skin.properties` or `skin_custom.properties` files can also contain skeleton path information that is used if skeleton attributes are omitted from the look & feel (`.laf`) file, as described in The Look & Feel File.

The following table shows an example of how entries in `skin.properties` for the active skin are converted to HTML `<head>` entries. Different skins may have different entries.

Table 2 How skin properties map to HTML entries

skin.properties Entries	Rendered HTML <head> Entries
images.path: images	content="/framework/skins/avitek/images"/>
link.body.href: css/body.css link.body.rel: stylesheet link.book.href: css/book.css link.book.rel: stylesheet link.button.href: css/button.css link.button.rel: stylesheet link.fix.href: css/fix.css link.fix.rel: stylesheet link.form.href: css/form.css link.form.rel: stylesheet link.layout.href: css/layout.css link.layout.rel: stylesheet link.portlet.href:	<link href="/sampleportal/framework/skins/avitek/css/body.css" rel="stylesheet"/> <link href="/sampleportal/framework/skins/avitek/css/book.css" rel="stylesheet"/> <link href="/sampleportal/framework/skins/avitek/css/button.css" rel="stylesheet"/> <link href="/sampleportal/framework/skins/avitek/css/fix.css" rel="stylesheet"/> <link href="/sampleportal/framework/skins/avitek/css/form.css" rel="stylesheet"/> <link href="/sampleportal/framework/skins/avitek/css/layout.css" rel="stylesheet"/> <link href="/sampleportal/framework/skins/avitek/css/portlet.css" rel="stylesheet"/> <link href="/sampleportal/framework/skins/avitek/css/window.css" rel="stylesheet"/>

css/portlet.css link.portlet.rel: stylesheet link.window.href: css/window.css link.window.rel: stylesheet link.window-plain.href: css/plain/window.css link.window-plain.rel: stylesheet	<link href="/sampleportal/framework/skins/avitek/css/plain/window.css" rel="stylesheet"/>
script.skin.src: skin.js script.skin.type: text/javascript script.menu.src: menu.js script.menu.type: text/javascript script.float.src: float.js script.float.type: text/javascript script.menufx.src: menufx.js script.menufx.type: text/javascript script.util.src: util.js script.util.type: text/javascript script.delete.src: delete.js script.delete.type: text/javascript script.search.path: js (Provides the directory for the location of the JavaScript files.)	<script type="text/javascript" src="/sampleportal/framework/skins/avitek/js/skin.js"></script> <script type="text/javascript" src="/sampleportal/framework/skins/avitek/js/menu.js"></script> <script type="text/javascript" src="/sampleportal/framework/skins/avitek/js/float.js"></script> <script type="text/javascript" src="/sampleportal/framework/skins/avitek/js/menufx.js"></script> <script type="text/javascript" src="/sampleportal/framework/skins/avitek/js/util.js"></script> <script type="text/javascript" src="/sampleportal/framework/skins/avitek/js/delete.js"></script>

You can control the order in which the CSS and JavaScript entries are inserted into the HTML `<head>` section by adding

`link.input.index:1`

to a CSS entry and

`script.util.index:1`

to a script entry, where the number is the order in which the entry should be inserted. All

CSS entries are inserted first, followed by all script entries.

Look & Feel Overrides

You can override the skin elements and skeletons on individual portal components so that those components have a different look & feel than the other portal components. For example, you can override the look & feel of a portlet so that it looks different than the other portlets on a page.

Overriding Look & Feel with Themes

As part of each skin or skeleton, you can create sub-skins and sub-skeletons called "themes." Themes contain all or part of the resources contained in a skin or skeleton. For example, a skin theme can contain a `/css` subdirectory with a single CSS file, and a skeleton theme can contain a single JSP to render a portlet titlebar. Themes can be used on books, pages, and portlets.

Each theme requires a `.theme` file located in `<portal_Web_project>/framework/markup/theme/`. Following is a sample theme file:

```
<?xml version="1.0" encoding="UTF-8"?>

<netuix:markupDefinition xmlns:netuix="http://www.bea.com/servers/netuix/xsd/con

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.bea.com/servers/netuix/xsd/controls/netuix/1.0.0

    <netuix:locale language="en"/>

    <netuix:markup>

        <netuix:theme

            name="alert"

            title="Alert Theme" description="A simple alert theme."

            markupType="Theme" markupName="alert"/>

    </netuix:markup>

</netuix:markupDefinition>
```

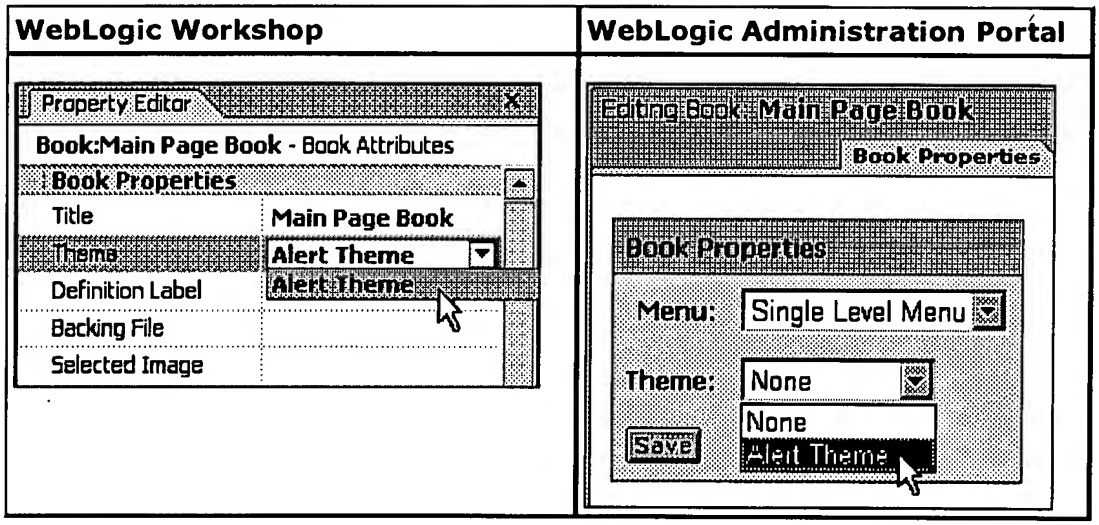
The theme file contains two key attributes:

- **name** - The name attribute value tells the portal framework the name of the theme directory to look in to apply theme resources to the book, page, or portlet.
- **title** - The title attribute value is used to populate the Theme drop-down list where it appears in the book, page, and portlet properties in the Portal Designer and in the WebLogic Administration Portal.

The theme XML is inserted in the `.portal` around the XML for the book, page, or portlet to

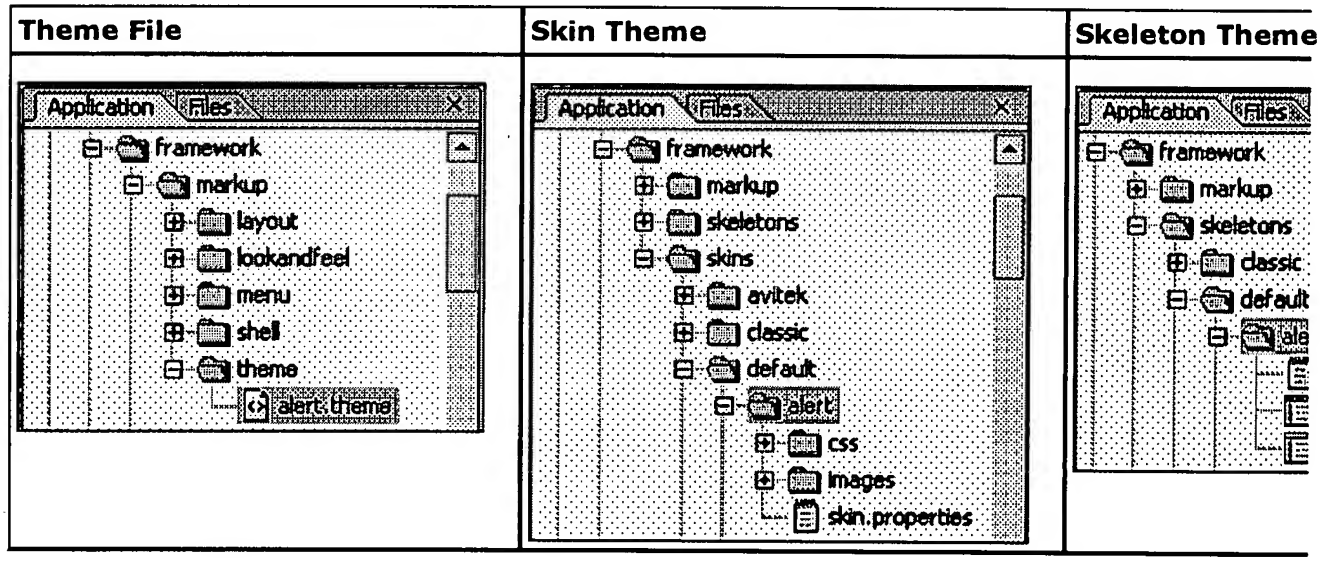
which the theme applies.

The following figures show where you set a theme in WebLogic Workshop and in the WebLogic Administration Portal.



Theme selection for a book, page, or portlet does not depend on the look & feel selected for a desktop. All themes are available for selection for all look & feels, whether or not the skins and skeletons for the look & feels contain the selected theme. If a skin or skeleton does not contain the selected theme, the theme is ignored. If both a skin and a skeleton theme exists for the selected look & feel, both are used.

The following figures show an example theme directory structure for the theme file, a skin theme, and a skeleton theme:



If skin or skeleton resources are not explicitly contained in a theme, the parent skin or skeleton resources are used. For example, if a skeleton theme uses only a JSP to render a portlet titlebar, the parent skeleton JSPs are used to render the rest of the portlet.

For skeletons, the ability to use parent resources is dependent on a file in the skeleton theme directory called `skeleton.properties`,which contains a single entry:

```
jsp.search.path: ., ..
```

where ., .. is a relative path to the theme's own skeleton JSPs and to the parent skeleton's JSPs.

In the parent skin, the `skin.properties` must contain path information to its skin themes in the following format:

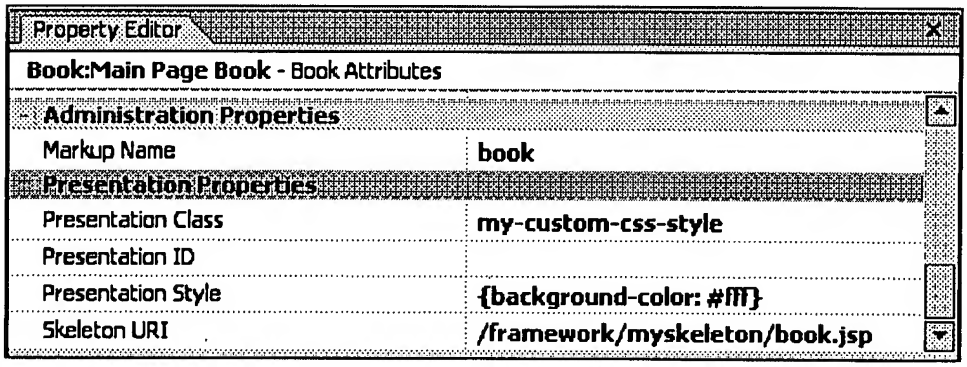
```
theme.alert.search.path: alert/images, images
```

The name of the theme directory is the second entry in the property. The path to the theme images is set (`alert/images`), along with the path to the parent skin's images directory (images) in case the theme images are an incomplete subset of the necessary images.

Overriding Look & Feel with Properties

For any selected component in the Portal Designer, you can override CSS properties and the skeleton JSP used to render the component. With the portal component selected in the Portal Designer, set the property overrides you want in the Property Editor under Presentation Properties, as shown in the following figure.

Figure 7 Property Editor



When the portal desktop is rendered as HTML, the skeleton JSP you selected is used to render the component, and the style overrides you entered are automatically inserted into the XML of the `.portal` file.

Summary

The look & feel selected for a portal desktop serves as the basis for how the desktop is rendered in HTML. The look & feel XML file (`.laf`) points to a specific skin and a specific skeleton on the file system to use for rendering.

Skins are made up of framework images (like portlet titlebar icons), CSS files, and script files, such as JavaScript. Skeletons are JSPs that convert XML-based portal components to HTML.

Once a look & feel is selected, its XML is inserted into the `.portal` XML file, which is the primary XML file used to control desktop rendering (`.portlet` XML files are used to render portlets). The look & feel settings point to the file-based skin and skeleton resources that are

used to generate and used in the rendered HTML.

The skin used in a look & feel contains a `skin.properties` and an optional `skin_custom.properties` file that contains references to all images, CSS files, and script files that are used by the skin. The entries in `skin.properties` and `skin_custom.properties` are converted to HTML `<head>` entries so that any framework images, CSS styles, and script functions used in the HTML are recognized.

You can override the look & feel for any book, page, or portlet by using themes; and using the Portal Designer and Portlet Designer Property Editor you can override CSS styles, attributes, and the skeleton JSP used to render desktops, books, pages, and portlet title bars and windows.

How the Shell Determines Header and Footer Content

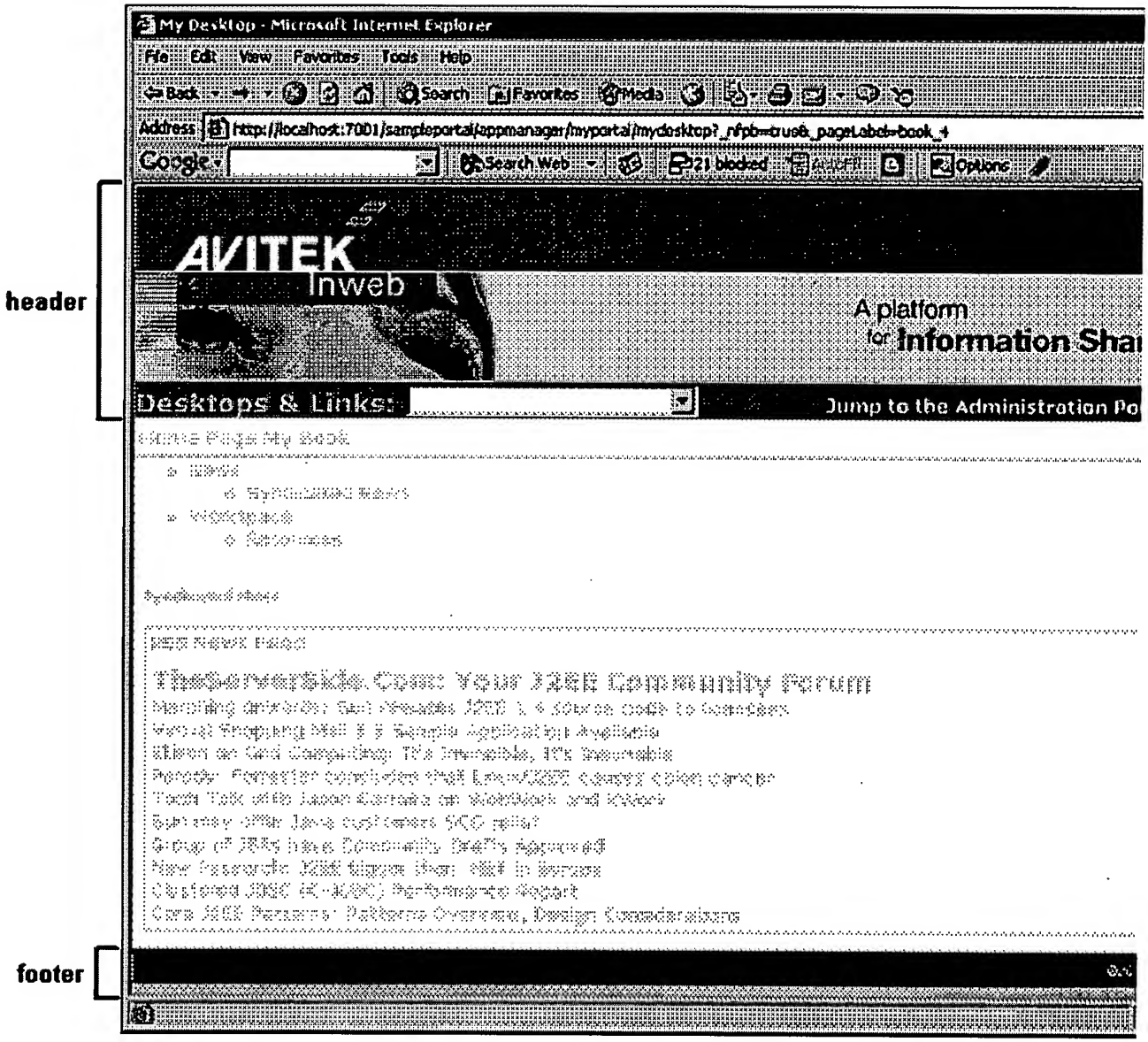
When you build a portal in WebLogic Workshop, the shell that you select determines the header and footer content of the portal desktop. The shell can point to JSP or HTML files that contain the content, personalization, or other behavior you want to include in your headers and footers.

This topic contains the following sections:

Overview

The following figure shows the area of a portal desktop controlled by the shell:

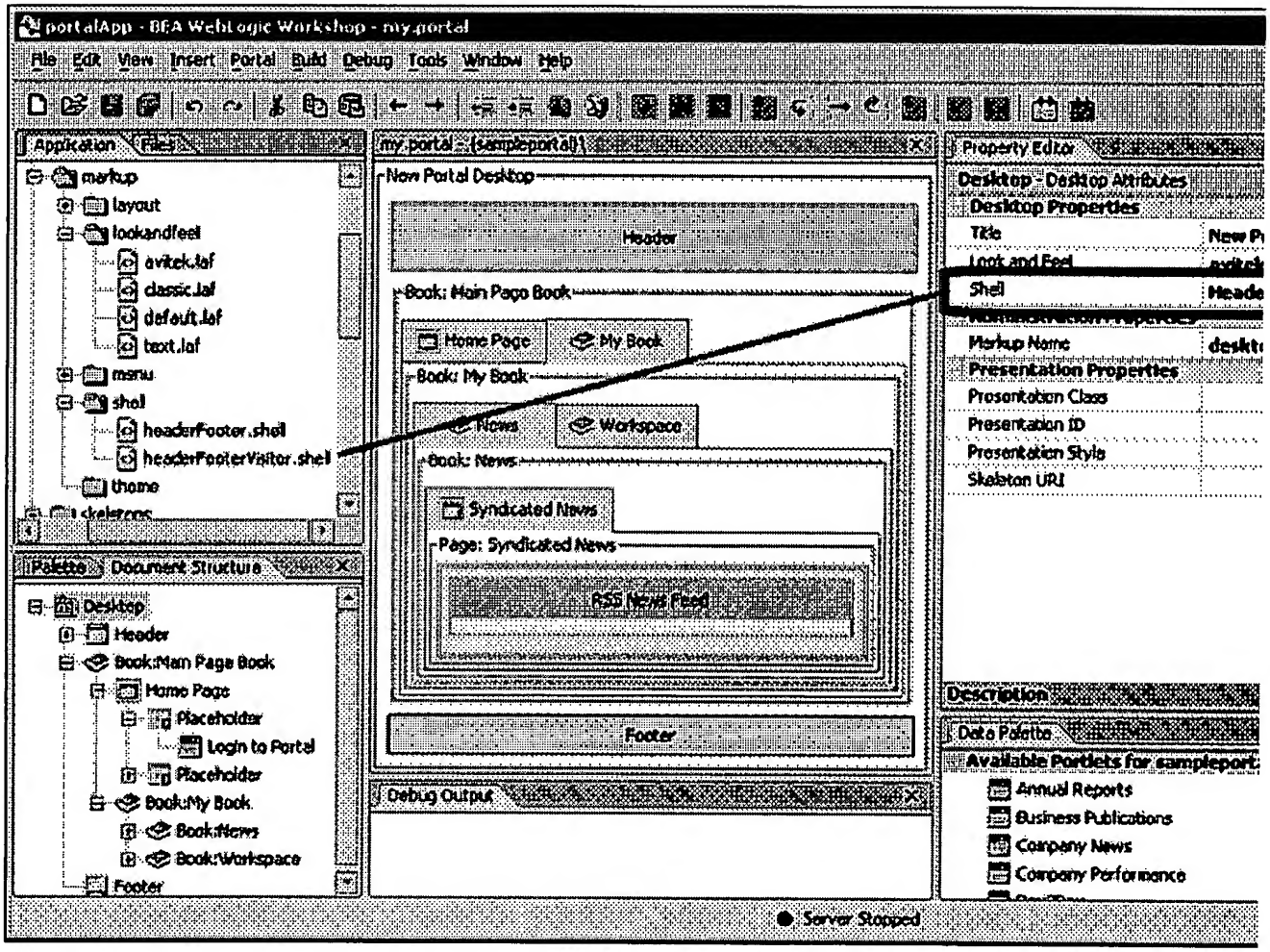
Figure 8 Portal areas controlled by the shell



The shell could also be set up to include a left navigation region, as illustrated in the left navigation sample in the WebLogic Workshop sample portal. So the shell really controls everything outside the main page book in a portal.

A shell is represented by an XML file (with `.shell` extension), as shown in the following figure.

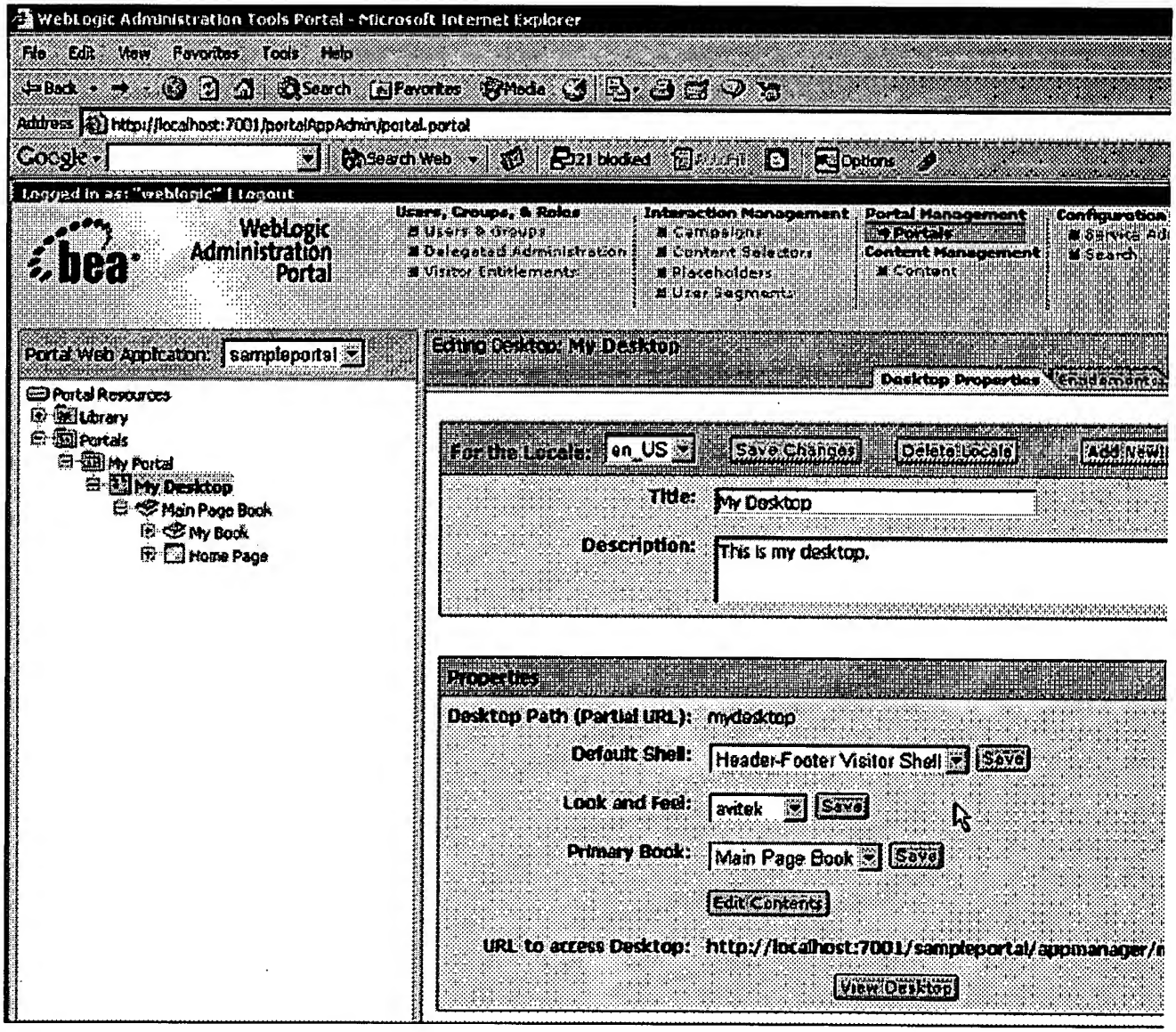
Figure 9 Shell XML file



Developers building portals with WebLogic Workshop are not the only users who can determine the shell used by a portal desktop. While developers create shells and select the default shell used by a portal, portal administrators ultimately determine the desktop shell.

After a portal administrator creates a desktop in the WebLogic Administration Portal, the administrator can change the desktop shell on the Desktop Properties page.

Figure 10 Changing the desktop shell



The following section shows the shell XML file and describes how it is used to provide header and footer content.

The Shell File

The shell provides paths to the JSP or HTML files to be used in the desktop header and footer.

Shell files are stored in the following location: `<portal_Web_project>/framework/markup/shell/`. Following is the `headerFooterVisitor.shell` provided by BEA with the key attributes highlighted.

```
<?xml version="1.0" encoding="UTF-8"?>

<netuix:markupDefinition xmlns:netuix="http://www.bea.com/servers/netuix/xsd/con

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xsi:schemaLocation="http://www.bea.com/servers/netuix/xsd/controls/netuix/1.0.0
    <netuix:locale language="en"/>
    <netuix:markup>
        <netuix:shell
            title="Header-Footer Visitor Shell"
            description="A header with a link and footer is included in this she
            markupType="Shell" markupName="headerFooterVisitor">
            <netuix:head/>
            <netuix:body>
                <netuix:header>
                    <netuix:jspContent contentUri="/portlets/header/header.jsp"/
                </netuix:header>
                    <netuix:break/>
                <netuix:footer>
                    <netuix:jspContent contentUri="/portlets/footer/footer.jsp"/
                </netuix:footer>
            </netuix:body>
        </netuix:shell>
    </netuix:markup>
</netuix:markupDefinition>
```

The following table describes the shell attributes and shows how they are used to put content in the desktop header and footer:

Table 3 Shell Attributes

This <element> or attribute...	does this
title	Required. The string used to display the name in the shell drop-down fields in WebLogic Workshop and the WebLogic Administration Portal.
description	Optional. Description of the shell. The description is used in the WebLogic Administration Portal; when you select a shell in the portal Library, the description appears on the Shell Properties

	page.
markupType	Required. The name of the type of component. Must always be "Shell".
markupName	Required. The name for the shell. Each shell in the portal Web project must have a unique `markupName`.
<netuix:head/>	Required. This element maps to the `head.jsp` skeleton file that renders the boundaries of the HTML `<head>` region.
<netuix:body>	Required. This element maps to the `body.jsp` skeleton file that renders the boundaries of the HTML `<body>` region.
<netuix:header>	Required. This element maps to the `header.jsp` skeleton file that renders the boundaries of the header region in HTML.
<netuix:footer>	Required. This element maps to the `footer.jsp` skeleton file that renders the boundaries of the footer region in HTML.
<netuix:jspContent>	Optional. Use this element to reference the JSPs or HTML files you want to use for content in the header and/or footer (by way of the `contentUri` attribute). To use this element, make sure the `<netuix:header>` and `<netuix:footer>` tags have opening and closing elements inside which this tag is inserted. Use the `contentUri` attribute to reference the JSP or HTML file relative to the portal Web project.

The shell XML is automatically added to the underlying XML in the `.portal` file.

The Portal File

Following is an example portal file, created with the Portal Designer, showing the inserted shell XML from the `.shell` file (in bold). The shell XML was inserted when the Shell property was set for the selected desktop in the Property Editor. When the `.shell` file is inserted into the `.portal` file, its job is finished in the rendering process and the `.portal` file is used to set the header and footer content.

```
<?xml version="1.0" encoding="UTF-8"?>

<portal:root xmlns:html="http://www.w3.org/1999/xhtml-netuix-modified/1.0.0"

    xmlns:netuix="http://www.bea.com/servers/netuix/xsd/controls/netuix/1.0.0"

    xmlns:portal="http://www.bea.com/servers/netuix/xsd/portal/support/1.0.0"

    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

    xsi:schemaLocation="http://www.bea.com/servers/netuix/xsd/portal/support/1.0

    <portal:directive.page contentType="text/html;charset=UTF-8"/>

    <netuix:desktop definitionLabel="defaultDesktopLabel" markupName="desktop" m
```

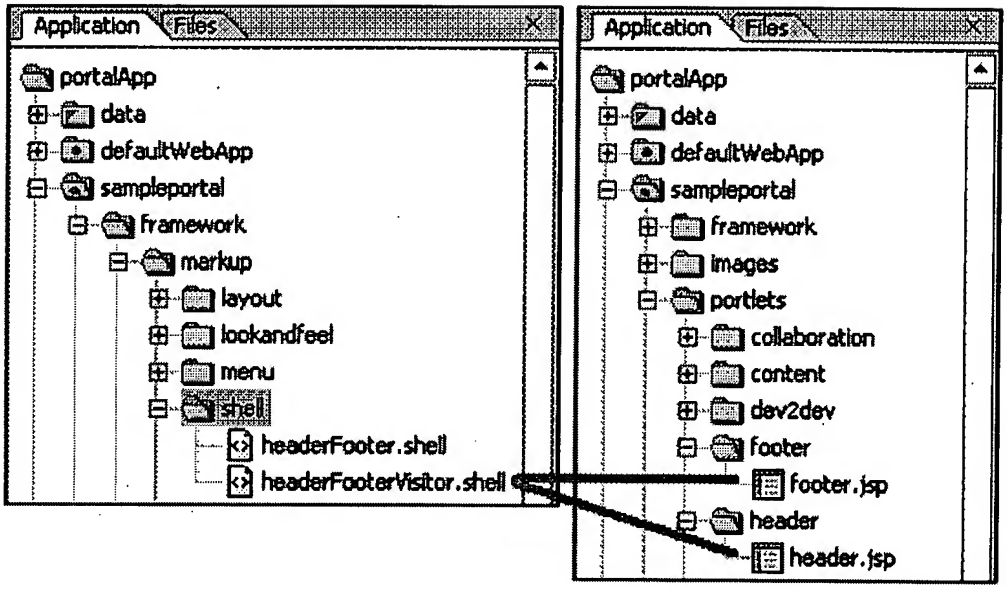
```
<netuix:lookAndFeel definitionLabel="avitek" description="The avitek loo
and feel"
    markupName="avitek" markupType="LookAndFeel" skeleton="default"
    skeletonPath="/framework/skeletons/" skin="avitek"
    skinPath="/framework/skins/" title="avitek"/>
<netuix:shell description="A header with a link and footer is included i
this shell."
    markupName="headerFooterVisitor" markupType="Shell" title="Header-F
    <netuix:head/>
    <netuix:body>
        <netuix:header>
            <netuix:jspContent contentUri="/portlets/header/header.jsp"/
        </netuix:header>

            [XML for books, pages, and portlets...]
        <netuix:footer>
            <netuix:jspContent contentUri="/portlets/footer/footer.jsp"/
        </netuix:footer>
```

The portal file is a template with which multiple desktops can be created in the WebLogic Administration Portal. When used as a template, the portal file determines the default shell of any desktop created from it.

Location of the Shell Resources

The shell attributes in the portal file tell the portal which content to use for the portal header and footer when the portal is rendered in HTML. The portal in the previous example will use the following shell resources:

Figure 11 Shell resources



The `headerFooterVisitor.shell` example file contains two tags that point to the content to use in the header and footer, one inside the `<header>` element and one inside the `<footer>` element:

```
<netuix:jspContent contentUri="/portlets/header/header.jsp"/>

<netuix:jspContent contentUri="/portlets/footer/footer.jsp"/>
```

The locations of those files are highlighted in the previous figure. When the portal is rendered, those JSPs are converted to HTML and inserted into the header and footer regions of the portal. The JSP files can contain any content or functionality allowed in a JSP, including personalization.

The JSPs referenced by this example shell do not have to be called `header.jsp` and `footer.jsp`. They could be any JSPs in the portal Web project. However, the skeleton JSPs used to render the boundaries of the header and footer regions are always called `header.jsp` and `footer.jsp` in the skeleton framework. The skeleton JSPs are different than the JSPs referenced by the shell. The following section explains the difference in more detail.

How the Shell Relates to Look & Feel

While the shell controls the content of the area surrounding the main book of a portal, the look & feel determines which skeleton `header.jsp` and `footer.jsp` are used to render the boundaries and styles of the header and footer areas.

In the following examples, do not be confused by the identically named `header.jsp` for both the look & feel and the shell header. They are different files with different uses. The fact that both have the same name is coincidence.

Look & Feel (header.jsp skeleton)

The "avitek" look & feel in the portal file uses the "default" skeleton located

in /sampleportal/framework/skeletons/default/. Included in the default skeleton is a file called header.jsp that is used to render the <header> element in the portal file.

```
<%@ page import="com.bea.netuix.servlets.controls.application.HeaderPresentation

<%@ page session="false"%>

<%@ taglib uri="render.tld" prefix="render" %>

<%

    HeaderPresentationContext header = HeaderPresentationContext.getHeaderPresen

%>

<render:beginRender>

    <%-- Begin Body Header --%>

    <div

        <render:writeAttribute name="id" value="<%= header.getPresentationId() %

        <render:writeAttribute name="class" value="<%= header.getPresentationCla

        <render:writeAttribute name="style" value="<%= header.getPresentationSty

    >

</render:beginRender>

[The JSP referenced in the shell <header> element is inserted here at rendering.

<render:endRender>

    </div>

    <%-- End Body Header --%>

</render:endRender>
```

This is a simple skeleton file that, when rendered, produces the following HTML:

```
<!-- Begin Body Header -->

<div

    class="bea-portal-body-header"

>

</div>
```

```
<!-- End Body Header -->
```

The opening tag uses a CSS class called bea-portal-body-header and then closes itself. The ending tag at the end of rendering closes the section. The JSP referenced in the shell header is inserted between the opening and closing tags where its content is rendered as shown in the following example:

```
<!-- Begin Body Header -->

<div

    class="bea-portal-body-header"

>

[The JSP referenced in the shell <header> element is inserted here at rendering.

</div>

<!-- End Body Header -->
```

The shell's header.jsp inserted into the tag of the header region controls the content, styles, and behavior of the header content. The only elements provided by the look & feel are the tag and the bea-portal-body-header style class.

For troubleshooting purposes, you could view the rendered portal and view the bea-portal-body-header class (contained in the avitek skin's body.css) to find out which style elements for which the look & feel is responsible. Following is the definition of bea-portal-body-header:

```
.bea-portal-body-header, .bea-portal-body-footer

{

    margin: 0px;

    padding: 1px;

    color: #C3C6B1;

}

.bea-portal-body-header

{

    font-size: large;

    font-weight: bold;

}
```

Summary

The shell selected for a portal desktop determines the content of the area surrounding the portal's main book. The shell XML file (`.shell`) includes references to the HTML or JSP files you want to appear in the desktop header and footer.

HTML and JSP files used in a header or footer can contain any content or functionality allowed in those types of files, including personalization in JSP files.

Once a shell is selected, its XML is inserted into the `.portal` XML file, which is the primary XML file used to control desktop rendering (`.portlet` XML files are used to render portlets).

While look & feel determines the physical boundaries of the header and footer and can include CSS styles and other skin elements generated by the skeleton `header.jsp` or `footer.jsp` files, the HTML or JSP files inserted in the header or footer by the shell control the content, styles, style overrides, and behavior of the header and footer.

How Portal Components Are Rendered

With the look & feel and shell selected for a portal desktop, the rendering service has the basic information it needs to convert a `.portal` XML file into a final HTML file.

This topic shows the rendering lifecycle, step by step, for a single portal component. The same rendering principles apply for all other portal components.

This topic includes the following sections:

- Overview
- Single File vs. Streamed Rendering
- Rendering Lifecycle of a Book
- Summary

Overview

There are three basic stages in the portal rendering process—a process that ultimately results in a portal desktop being displayed in a browser:

1. Building a portal in XML: In the portal development process, you use the Portal and Portlet designers in WebLogic Workshop to build `.portal` and `.portlet` files. Both types are XML files. As you build portals and portlets in WebLogic Workshop, the XML elements and attributes are automatically built under the surface.

The previous topics, How Look & Feel Determines Rendering and How the Shell Determines Header and Footer Content, described part of the XML-building process: how the look & feel and shell XML files are added to the portal XML file to provide rendering instructions.

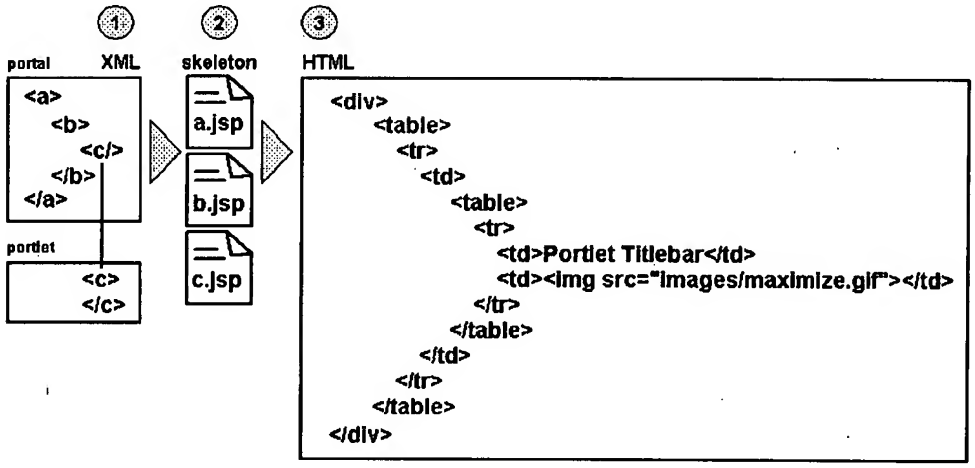
2. Portal XML elements mapped to JSP skeleton files: The portal framework maps specific XML elements to specific JSP skeleton files. They are called skeleton files because they are used to render the physical boundaries and structure—the skeleton—of their portal components. For example, a portlet titlebar in a portlet XML file uses an element called

<netuix:titlebar>. The portal framework knows to use the `titlebar.jsp` skeleton file to render the portlet titlebar.

3. JSP skeleton files and skin.properties are rendered as HTML: Each skeleton JSP file performs its own processing, such as retrieving property values you set in the WebLogic Workshop Property Editor (and were automatically added to the portal XML file), and generates the appropriate HTML for the portal component. The `skin.properties` and optional `skin_custom.properties` files for the selected look & feel are converted to image path entries, CSS file entries, and script file entries in the HTML <head> area.

The following figure is a simplified illustration of the rendering process.

Figure 12 Portal rendering process



This topic will expand on these three stages using the rendering lifecycle of a single portal component as an example.

Before going into greater detail on the rendering process, it is important to understand the difference between viewing a portal in the development environment (WebLogic Workshop) and viewing it in the administration/end user environment (WebLogic Administration Portal/browser). The three-stage rendering process occurs in slightly different ways in the two different environments. The following section describes the basic principles of each.

Single File vs. Streamed Rendering

The `.portal` file you create in WebLogic Workshop is a template. In this template you create books, pages and portlets and define defaults for them. When you view the `.portal` file with your browser the portal is rendered in "single file mode," meaning that you are viewing the portal from your file system as opposed to a database. The `.portal` file's XML is parsed and the rendered portal is returned to the browser. The creation and use of a `.portal` is intended for development purposes, but you can access a `.portal` file in production. Because there is no database involved you cannot take advantage of features such as user customization or entitlements.

Once you have created a `.portal` file you can use it to create desktops for a production environment.

A desktop is a particular view of a portal that visitors access. A portal can be made up of

multiple desktops, making the portal a container for desktops. A desktop contains all the portlets, content, shells, layouts, and look & feel elements necessary to create individual user views of a portal.

When you create a desktop based on the `.portal` file in the WebLogic Administration Portal, the `.portal` and its resources are placed into the database. The settings in the `.portal` file, such as the look & feel, serve as defaults to the desktop. Once a new desktop is created from a `.portal` template, the desktop is decoupled from the template, and modifications to the `.portal` file do not affect the desktop, and vice versa. For example, when you change a desktop's look & feel in the WebLogic Administration Portal, the change is made only to the desktop, not to the original `.portal` file. When you view a desktop with a browser it is rendered in "streaming mode" (from the database). Now that a database is involved, desktop customizations can be saved and delegated administration and entitlements can be set on portal resources.

The following table compares streamed and file based portals in more detail:

Portal Feature	**File-based Portals (.portal XML file)**	**Streamed (database generated) Portals**
Adding Entitlements	Run-time check only	Yes—More easily set and configured
Setting Preferences Number of Instances	In portal definition Limited No	For individual portal instances More than file-based portals Yes
Customization	No	Yes (Through Visitor Tools and the Admin Portal)
Internationalization	Difficult—Requires changes to skeleton files.	Easier
Performance	Slight advantage	Slightly less than file-based portals
Propagation (from test to production environments)	Easy to accomplish by moving the `.portal` file	More difficult. Requires utilities and proper planning.
Development Process	Easiest	More difficult

Rendering Lifecycle of a Book

This section illustrates the rendering lifecycle of a book, which will help you understand the rendering lifecycle of other portal components, such as pages and portlets.

This section contains the following topics:

1. Building a portal in XML
2. Portal XML elements mapped to JSP skeleton files
3. JSP skeleton files and `skin.properties` are rendered as HTML

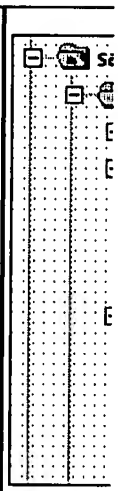
1. Building a portal in XML

This section describes steps that populate and configure the `.portal` XML file.

Selecting Look & Feel

When you select the look & feel for a desktop, the look & feel file determines which skin and skeleton is used to render all desktop components. In the following example, the "avitek" look & feel has been selected, which uses the "avitek" skin and the "default" skeleton. The look & feel XML is added to the `.portal` XML file.

```
<netuix:markup>

    <netuix:lookAndFeel

        definitionLabel="avitek" title="avitek"

        description="The avitek look and feel"

        skin="avitek" skinPath="/framework/skins/"

        skeleton="default" skeletonPath="/framework/skeletons/"

        markupType="LookAndFeel" markupName="avitek"/>

</netuix:markup>
```



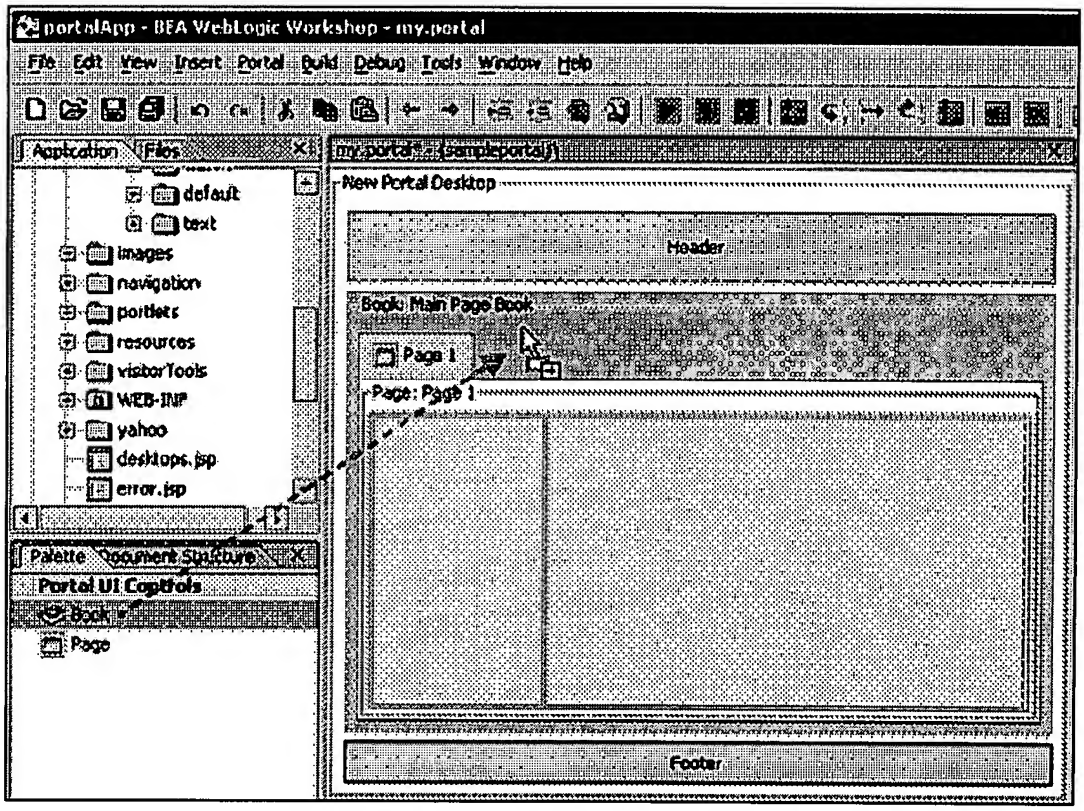
The skin and skeleton come into play later in the rendering process, when the desktop is viewed with a browser. Before that happens, the book that will be used to illustrate the rendering process will be added to the portal.

Adding a Book to a Portal

In this section a book is added to the desktop in the `.portal` file and configured. Books can also be added by portal administrators in the WebLogic Administration Portal, which adds the book directly to the database.

The following figure shows a book control being dragged onto the desktop.

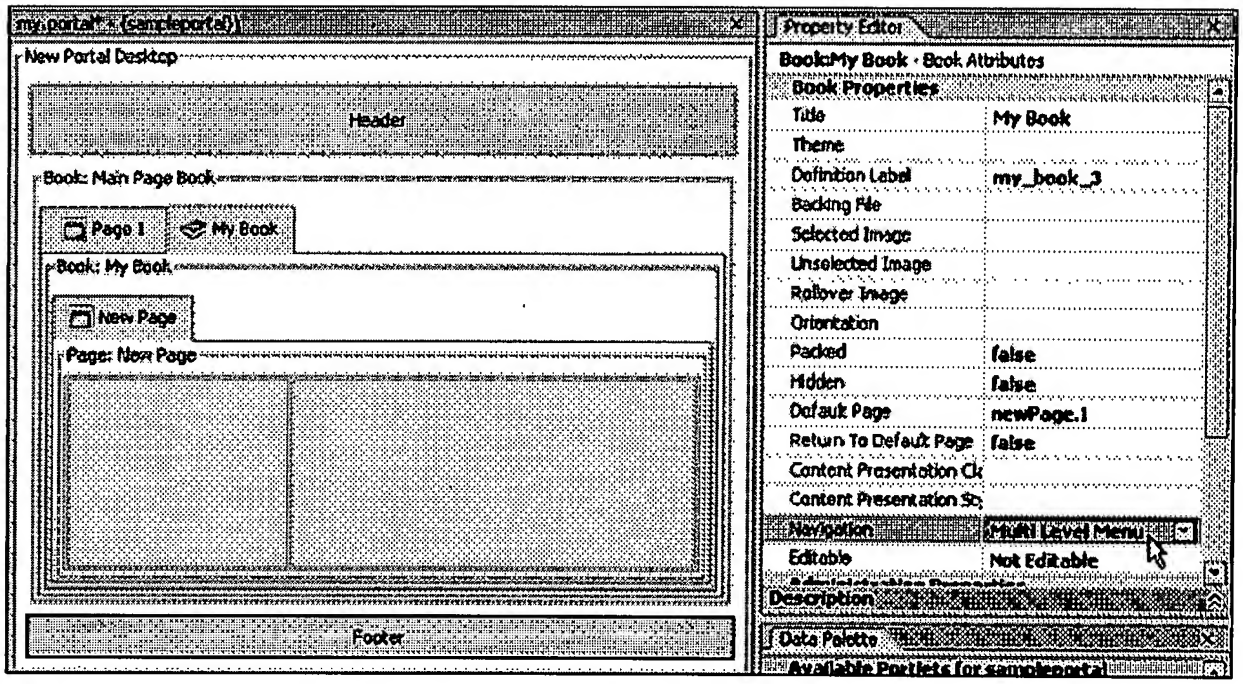
Figure 13 Dragging a control to the Desktop



After the book is added to the desktop, the book title is changed from "New Book" to "My Book," and the navigation style is set to Multi Level Menu, as shown in the following figure.

Navigation controls the way a book's sub-books and pages are accessed. The single-level menu provides text links/tabs to sub-books and pages, and the multi-level menu provides a drop-down menu to access sub-books and pages. (Books must be added to books rather than to pages inside books for drop-down navigation to work. So in the following example, for the multi-level menu to produce a drop-down menu, you would need to drag a new book control into Main Page Book, right next to Page 1, as shown in the following figure.)

Figure 14 Arranging portal components



After the Navigation style is set on My Book, the following is what the book looks like in XML in the `.portal` file. If you add a book in the WebLogic Administration Portal, the XML is added to the database. This XML is used as the basis for the rendering of the book.

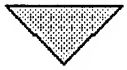
```
<netuix:book defaultPage="newPage.1" definitionLabel="my_book_3"
    markupName="book" markupType="Book" title="My Book">
    <netuix:multiLevelMenu
        description="This menu can navigate across may nested books."
        markupName="multiLevelMenu" markupType="Menu" title="Multi Level Menu"/>
            <!-- in this example, the nested page content has been removed -->
</netuix:book>
```

2. Portal XML elements mapped to JSP skeleton files

When the desktop is viewed in a browser, the portal framework reads the XML elements and uses the skeleton path to map the desktop's XML elements to skeleton JSPs. The following examples show which elements in the book XML are mapped to skeleton JSPs and which skeleton JSPs are used to render the elements.

```
book XML - The highlighted elements are mapped to skeleton JSPs.

<netuix:book defaultPage="newPage.1" definitionLabel="my_book_3"
```

```
        markupName="book" markupType="Book" title="My Book">

        <netuix:multiLevelMenu

            description="This menu can navigate across may nested books."

            markupName="multiLevelMenu" markupType="Menu" title="Multi Level Men

                <!-- in this example, the nested page content has been removed -

    </netuix:book>
```



```
    skeleton - Referenced in the look & feel

    /framework/skeletons/default/
    book.jsp
    multilevelmenu.jsp
    submenu.jsp (referenced in multilevelmenu.jsp)
```

Once rendering has been handed off to the JSPs, the JSPs perform the tasks necessary for conversion to HTML. Following are the `book.jsp`, `multilevelmenu.jsp`, and `submenu.jsp` used in this example. Comments are added to describe what the JSPs are doing.

book.jsp

The `book.jsp` serves as a high-level container for the book's menu and the book's child books and pages. Comments in the JSP code are highlighted in bold text.

```
<%@ page import="com.bea.netuix.servlets.controls.page.BookPresentationContext,
                 com.bea.netuix.servlets.controls.page.MenuPresentationContext"
%>
<%@ page session="false"%>
<%@ taglib uri="render.tld" prefix="render" %>

<render:beginRender>


<%-- The content inside the <render:beginRender> tag is processed first

     and ultimately renders whatever is inside it. In most cases, the
```

```
      skeletons produce an opening <div> HTML tag with specific attributes
      such as CSS classes.
      The following block determines where the book falls in the desktop
      hierarchy (whether it is the top-level book or a nested book). It
      also sets the base name of the CSS class to use (bea-portal-book)
      and appends different endings to the base class to apply a
      different CSS class for each book context. Only processing,
      not HTML rendering, occurs in this block.
--%>

<%
    BookPresentationContext book =
          BookPresentationContext.getBookPresentationContext(request);
    MenuPresentationContext menu = (MenuPresentationContext)
          book.getFirstChild("page:menu");
    String bookClass = "bea-portal-book";
    String useBookClass = bookClass;

    if (book.isDesktopBook())
    {
        bookClass += "-primary";
        useBookClass = bookClass;
    }
    else if (book.isLikePage())
    {
        useBookClass += "-invisible";
    }

    String bookContentClass = bookClass + "-content";
%>

<%-- The next block begins the actual HTML rendering, beginning with
      the comment "Begin Book" followed by an opening <div> HTML
      tag. Notice the JSP tags used before the closing bracket of the
      <div> tag. These populate the div tag with style attributes.
      The methods retrieve any presentation property override values
      you entered in the WebLogic Workshop Property Editor for the book.
      For the "class" attribute, the default value is useBookClass,
```

```
      which earlier is set to "bea-portal-book". (If through getting
      the context the book was found to be the top-level book, the value
      of useBookClass would be "bea-portal-book-primary".)
      With no overrides, the useBookClass variable will produce the
      following HTML, because the book is acting like a page:
      <div
          class="bea-portal-book-invisible"
      >
      The style sheet class is provided by the skin, and the CSS file
      containing the class is referenced in the skin's skin.properties
      file and added to the HTML <head> region.
  --%>
    <%-- Begin Book --%>
    <div
        <render:writeAttribute name="id" value="<%= book.getPresentationId() %>"
        <render:writeAttribute name="class" value="<%=
                    )
             book.getPresentationClass()%>" defaultValue="<%= useBookClass %>"/>
        <render:writeAttribute name="style" value="<%= book.getPresentationStyle
    >
  <%-- The following JSP tag gets the names of the pages and books it will
      display in its navigation menu, and based on the navigation element
      used in the portal XML file (in this case netuix:multiLevelMenu),
      uses the corresponding menu JSP (multilevelmenu.jsp) to render the menu
      in this position of the HTML.
  --%>
        <render:renderChild presentationContext="<%= menu %>"/>
  <%-- The following block provides a <div> HTML container for the
      book's content area--the child books and pages. Again, it uses
```

```
     a JSP tag to set the style sheet "class" attribute.

--%>

        <%-- Begin Book Content --%>
        <div
            <render:writeAttribute name="class" value="<%= book.getContentPrese
            <render:writeAttribute name="style" value="<%= book.getContentPresen
        >
</render:beginRender>

<%-- The closing </render:beginRender> tag signals the portal framework

     to stop rendering the book. After the book's children and their

     children are rendered, the portal framework uses the following

     <render:endRender> tag to close the book's parent HTML tags.

--%>

<render:endRender>
        </div>
        <%-- End Book Content --%>
    </div>
    <%-- End Book --%>
</render:endRender>
```

Following is a description of the `multilevelmenu.jsp`, which is used by the book to render the navigation menu for the book's child books and pages.

multilevelmenu.jsp

The `multilevelmenu.jsp` is rendered inside the book container and provides the boundaries for multi-level menus on books. This JSP also uses `submenu.jsp` to perform the actual rendering of the menu links.Comments are highlighted in bold text.

```
<%@ page import="com.bea.netuix.servlets.controls.window.WindowPresentationConte
                 com.bea.netuix.servlets.controls.page.BookPresentationContext,
                 com.bea.netuix.servlets.controls.page.MenuPresentationContext,
                 java.util.List,
                 java.util.Iterator,
                 com.bea.netuix.servlets.controls.page.PagePresentationContext,
                 com.bea.netuix.servlets.controls.window.WindowCapabilities" %>
<%@ page session="false"%>
<%@ taglib uri="render.tld" prefix="render" %>

<%-- The following block determines where the book falls in the desktop

     hierarchy (whether it is the top-level book or a nested book). It

     also sets the base name of the CSS class to use (bea-portal-book)

     and defines different menu style classes by appending different
```

```
     endings to the base class. Only processing, not HTML rendering,

     occurs in this block.

--%>

<%
    BookPresentationContext book =

        BookPresentationContext.getBookPresentationContext(request);
    MenuPresentationContext menu =

        MenuPresentationContext.getMenuPresentationContext(request);
    String bookClass = "bea-portal-book";

    if (book.isDesktopBook())
    {
        bookClass += "-primary";
    }

    final String menuClass = bookClass + "-menu";
    final String menuContainerClass = menuClass + "-container";
    final String menuItemClass = menuClass + "-item";
    final String menuItemActiveClass = menuItemClass + "-active";
    final String menuItemLinkClass = menuItemClass + "-link";
    final String menuHookClass = menuClass + "-hook";
    final String menuButtonsClass = menuItemClass + "-buttons";
    List menuChildren = menu.getChildren();
%>

<render:beginRender>

<%-- The content inside the <render:beginRender> tag is processed first

     and ultimately renders whatever is inside it, such as opening <div>

     HTML tags with specific attributes and tables.

     The following block creates a table cell, sets CSS styles on the <td>

     tag (based on the members defined in the previous block).

--%>

<%-- Begin Multi Level Menu --%>
    <div class="bea-portal-ie-table-buffer-div">
        <table border="0" cellpadding="0" cellspacing="0" width="100%">
            <tr>
                <td class="<%= menuContainerClass %>" align="left" nowrap="nowra

<%-- The following block builds the menu in the table cell. It first adds an

     unordered list <ul> to the cell and sets its style class. Then, an IF

     statement checks to see if the book is in VIEW mode. If true, CSS styles
```

are put in the request as attributes to be used by the menu.

After the attributes are added to the request, the skeleton's submenu.jsp is inserted, which does the following:

* Gets the CSS styles from the request.
* Gets the book's child pages and books.
* Creates list items <li> of the children and creates links out of them. The menuHookClass at the end of the block is used by the skin's menu.j to insert the rendered menu. The <ul> that is generated is a menu structure description that is read and rewritten by menu.js.
* Adds CSS styles to the request and includes submenu.jsp to handle the menus of nested books.

After the menu is built, the CSS styles are removed from the request.

```
--%>

                    <ul
                        <render:writeAttribute name="id" value="<%=

                         menu.getPresentationId() %>"/>
                        <render:writeAttribute name="class" value="<%=

                           menu.getPresentationClass() %>" defaultValue="<%=

                         menuClass %>"/>
                        <render:writeAttribute name="style" value="<%=

                         menu.getPresentationStyle() %>"/>
                    ><%
                        if (book.getWindowMode().equals(WindowCapabilities.VIEW)
                        {
                            request.setAttribute(BookPresentationContext.class.g
                            request.setAttribute(BookPresentationContext.class.g
                            request.setAttribute(BookPresentationContext.class.g
                            request.setAttribute(BookPresentationContext.class.g
                            request.setAttribute(BookPresentationContext.class.g
                            request.setAttribute(BookPresentationContext.class.g
                            %><jsp:include page="submenu.jsp"/><%
                            request.removeAttribute(BookPresentationContext.clas
                            request.removeAttribute(BookPresentationContext.clas
                            request.removeAttribute(BookPresentationContext.clas
                            request.removeAttribute(BookPresentationContext.clas
                            request.removeAttribute(BookPresentationContext.clas
                            request.removeAttribute(BookPresentationContext.clas
                        }
```

```
                    %></ul>

                    <div class="<%= menuHookClass %>"></div>
                </td>

<%-- The following block adds a table cell next to the menu table cell

     if a menu is present. The <render:endRender> contents are inserted

     in the HTML after all menu children are inserted, which closes

     the menu table.

--%>

<%
    if (menuChildren != null && menuChildren.size() > 0)
    {
%>
                <td class="<%= menuButtonsClass %>" align="right" nowrap="nowrap
<%
    }
%>
</render:beginRender>
<render:endRender>
<%
    if (menuChildren != null && menuChildren.size() > 0)
    {
%>
                </td>
<%
    }
%>
            </tr>
        </table>
    </div>
    <%-- End Multi Level Menu --%>
</render:endRender>
```

submenu.jsp

The submenu.jsp is inserted inside the multilevelmenu.jsp. It retrieves a book's child books and pages and builds the navigation links to those children. Comments are shown in bold text.

```
<%@ page import="java.util.Iterator,
                 java.util.List,
                 com.bea.netuix.servlets.controls.page.BookPresentationContext,
                 com.bea.portlet.PageURL,
                 com.bea.netuix.servlets.controls.page.PagePresentationContext"%
<%@ page session="false"%>

<%-- The following block gets the CSS styles placed in the request by

     multilevelmenu.jsp.
```

```
--%>

<%
    Boolean isRoot
        = (Boolean) request.getAttribute(BookPresentationContext.class.getName()
    BookPresentationContext bookCtx
        = (BookPresentationContext)

            request.getAttribute(BookPresentationContext.class.getName() +

         ".menu-item");
    String menuClass
        = (String) request.getAttribute(BookPresentationContext.class.getName()

         ".menu-class");
    String menuItemClass
        = (String) request.getAttribute(BookPresentationContext.class.getName()

         ".menu-item-class");
    String menuItemActiveClass
        = (String) request.getAttribute(BookPresentationContext.class.getName()

         ".menu-item-active-class");
    String menuItemLinkClass
        = (String) request.getAttribute(BookPresentationContext.class.getName()

         ".menu-item-link-class");
%>

<%-- The following block checks to see if the book and its children

     are visible. If true, the labels of the children are retrieved,

     iterated over, and inserted as hyperlinked list items <li>

     inside the unordered list <ul> inserted by multilevelmenu.jsp.

     Notice the nested <ul> at the end of the block, which provides

     for submenu nesting.

--%>

<%
    if (!bookCtx.isHidden() && bookCtx.isVisible())
    {
        if (bookCtx instanceof BookPresentationContext)
        {
            List bookChildren = bookCtx.getPagePresentationContexts();
            Iterator it = bookChildren.iterator();

            while (it.hasNext())
            {
                PagePresentationContext childPageCtx = (PagePresentationContext)
```

```
                    it.next();

                if (!childPageCtx.isHidden() && childPageCtx.isVisible())
                {

                  %><li class="<%= isRoot.booleanValue() &&

                   childPageCtx.isActive() ? menuItemActiveClass : menuItemClass

                  %>"><%
                    %><a class="<%= menuItemLinkClass %>" href="<%= PageURL.crea

                    if (childPageCtx instanceof BookPresentationContext)
                    {
                        %><ul class="<%= menuClass %>"><%
                        request.setAttribute(BookPresentationContext.class.getNa

                        + ".root-flag", Boolean.FALSE);
                        request.setAttribute(BookPresentationContext.class.getNa

                        + ".menu-item", childPageCtx);
                        %><jsp:include page="submenu.jsp"/><%
                        request.removeAttribute(BookPresentationContext.class.ge
                        request.removeAttribute(BookPresentationContext.class.ge
                        %></ul><%
                    }

                    %></li><%
                }
            }
        }
    }
%>
```

JavaScript in Menus

The menus in a desktop use JavaScript functions for such functionality as drop-down menus and rollovers. These JavaScript functions are called from the skeleton's `body.jsp`, which contains the following entry:

```
<render:writeAttribute name="onload" value="<%= body.getOnloadScript() %>"/>
```

The onload value is retrieved from the following property in the skin's `skin.properties` file:

```
document.body.onload: initSkin()
```

Following is the HTML written by the `body.jsp`:

```
<body

    class="bea-portal-body"

    onload="initSkin();"

>
```

The initSkin() JavaScript function is the base function that calls menu-rendering functions in

other JavaScript files. The initSkin() function is contained in the `skin.js` file. Other menu functions are contained in the `menu.js` and `menufx.js` files. Since all of those JavaScript files are listed in the skin's `skin.properties` file, they are automatically added to the HTML `<head>` region at rendering, and the functions they contain are recognized.

The next section describes the final process of the skeleton JSPs and `skin.properties` being converted to HTML.

3. JSP skeleton files and skin.properties are rendered as HTML

The previous section described the skeleton JSPs that are used to convert a book with a multi-level menu to HTML. The descriptions in that section described briefly some of the HTML generated by the JSPs.

This section shows the final HTML that is generated for a book, describes where it came from, and shows where some of the CSS styles used are defined.

Not all HTML for the desktop is shown in the following table. Only the sections that relate to the look & feel and the example book are shown.

skin.properties and **skin_custom.properties** - The paths to skeletons, skins, images, style sheets, and JavaScript files in the HTML `<head>` region are inserted from the skin's `skin.properties` and `skin_custom.properties` files. To see the original `skin.properties` entries, see The skin.properties File in "How Look & Feel Determines Rendering." The `<head>` tag is inserted by the `head.jsp` file used for the shell. The `<title>` is inserted from the desktop title in the `.portal` file.

The first three `<meta>` tags are for testing and debugging purposes. These can be removed from `skin.properties` by setting the `enable.meta.info` property to false.

```
<head>

<title>New Portal Desktop</title>

<meta name="bea-portal-meta-skeleton" content="/framework/skeletons/default"/>

<meta name="bea-portal-meta-skin" content="/framework/skins/avitek"/>

<meta name="bea-portal-meta-skin-images" content="/framework/skins/avitek/images

<link href="/sampleportal/framework/skins/avitek/css/body.css" rel="stylesheet"/

<link href="/sampleportal/framework/skins/avitek/css/button.css" rel="stylesheet

<link href="/sampleportal/framework/skins/avitek/css/window.css" rel="stylesheet

<link href="/sampleportal/framework/skins/avitek/css/plain/window.css" rel="styl

<link href="/sampleportal/framework/skins/avitek/css/portlet.css" rel="styleshee

<link href="/sampleportal/framework/skins/avitek/css/book.css" rel="stylesheet"/

<link href="/sampleportal/framework/skins/avitek/css/fix.css" rel="stylesheet"/>

<link href="/sampleportal/framework/skins/avitek/css/layout.css" rel="stylesheet
```

```
<link href="/sampleportal/framework/skins/avitek/css/form.css" rel="stylesheet"/

<script type="text/javascript" src="/sampleportal/framework/skins/avitek/js/menu

<script type="text/javascript" src="/sampleportal/framework/skins/avitek/js/util

<script type="text/javascript" src="/sampleportal/framework/skins/avitek/js/dele

<script type="text/javascript" src="/sampleportal/framework/skins/avitek/js/floa

<script type="text/javascript" src="/sampleportal/framework/skins/avitek/js/menu

<script type="text/javascript" src="/sampleportal/framework/skins/avitek/js/skin

</head>
```

The following section shows the HTML that is produced by each skeleton JSP.

book.jsp

```
<div

    class="bea-portal-book-invisible"

>
```

multilevelmenu.jsp

```
    <div class="bea-portal-ie-table-buffer-div">
        <table border="0" cellpadding="0" cellspacing="0" width="100%">
            <tr>
                <td class="bea-portal-book-menu-container" align="left"

                  nowrap="nowrap">
                    <ul
                        class="bea-portal-book-menu"
                    >
```

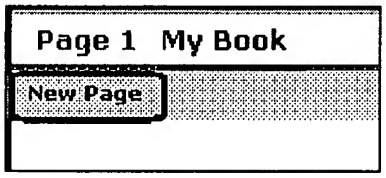
submenu.jsp

```
<li class="bea-portal-book-menu-item-active"><a class="bea-portal-book-menu-ite

href="http://localhost:7001/sampleportal/my.portal?_nfpb=true&_pageLabel=my_page

        <div
            class="bea-portal-book-content"
        >

        <%-- The book content (sub-books and pages) is inserted here. -->

        </div>
```

```
</div>                    </ul>
                      <div class="bea-portal-book-menu-hook"></div>
                  </td>

              </tr>

          </table>

      </div>
```

When the desktop for this example is rendered, the following appears in the browser:

Figure 15 Rendered desktop



The circled area in this figure is the only content rendered for the book. The book contains only one page, so there is only one menu item for the book. The "Page 1" and "My Book" tabs are menu items rendered by the parent Main Page Book. That is why you do not see the "My Book" in the previous HTML block: because the book is responsible for rendering only a menu of its child books and pages.

If "New Page" contained a portlet, the portlet would appear in the browser. However, the rendering of the page and portlet is handled by different skeleton JSPs: one to provide a container for the page content, one to render the layout of the page (table cells that contain portlets and sub-books), and a few to handle the rendering of the portlet.

The book is responsible for rendering only two things:

- The menu of sub-books and pages it contains.
- Opening and closing `<div>` tags to serve as the container for sub-books, pages, portlets, and other sub-components contained in the book.

CSS Styles in the Example

As you can see from the previous example of rendered HTML code, the skeleton JSPs insert many CSS styles. For example, the `multilevelmenu.jsp` inserts

```
<td class="bea-portal-book-menu-container" ...>
```

The style classes inserted by `multilevelmenu.jsp` are rewritten by the skin's `menu.js` file.

Also, some of the style classes inserted by the skeleton JSPs are not defined in any of the CSS files provided by BEA. You can add these style classes to your custom CSS files to control those styles in your portal desktops.

To determine which styles you want to modify, see The Look & Feel Editor.

Changing Look & Feel

If the look & feel is changed, a different skin and skeleton is referenced by the look & feel file, and rendering is subject to that skin and skeleton. With a different skin and skeleton, CSS files and script code can change completely.

Summary

There are three basic stages in the portal rendering process: building a portal in XML, portal XML components being mapped to skeleton JSPs, and skeleton JSPs rendering the portal desktop in HTML. The latter two stages are handled automatically by the portal framework.

There is a rendering difference between viewing a portal desktop in development mode and in administration/end user mode. In development mode, when you view the `.portal` file in a browser you see it in "single file" mode, meaning the desktop is being rendered from the file system. In administration/end user mode, you view a portal desktop in a browser in "streamed" mode, meaning the desktop components are being streamed from a database. When you create a portal desktop in the WebLogic Administration Portal using a `.portal` file as a template for the desktop, the portal components are added to the database and are decoupled from the original `.portal` file.

The Look & Feel Editor

The Look & Feel Editor lets you interactively edit the text styles used by portal text elements. Technically, the editor modifies Cascading Style Sheet (CSS) files that are referenced by a portal's `skin.properties` file. For example, using the Look & Feel Editor, you can change the size of a heading, the color of a list element, or the padding around a table cell for a portal.

The Look & Feel Editor also lets you change the properties of a portal's look & feel file (`.laf` file), such as the skin and skeleton files that it references.

In addition, the Editor shows you, at a glance:

- The CSS cascade for a portal
- The properties assigned to a selected CSS style
- The inherited properties of a selected CSS style
- The elements of the portal's `skin.properties` file

This topic discusses the functional parts of the Look & Feel Editor in greater detail than the corresponding online help topic. The goal is to offer additional insight into the purpose and use of the Editor than is covered elsewhere. This topic includes the following sections:

- Overview
- Application Window
- Style Hierarchy Window
- Style Description Window

- View Area
- Document Structure Window
- Property Editor Window
- Summary

Note: To use the Look & Feel Editor successfully, you must have a basic understanding of CSS. In this document, we provide minimal explanations of key CSS features, such as inheritance. If you are new to CSS, we recommend reviewing a book that covers the subject in detail. Many books and Websites are devoted exclusively to CSS.

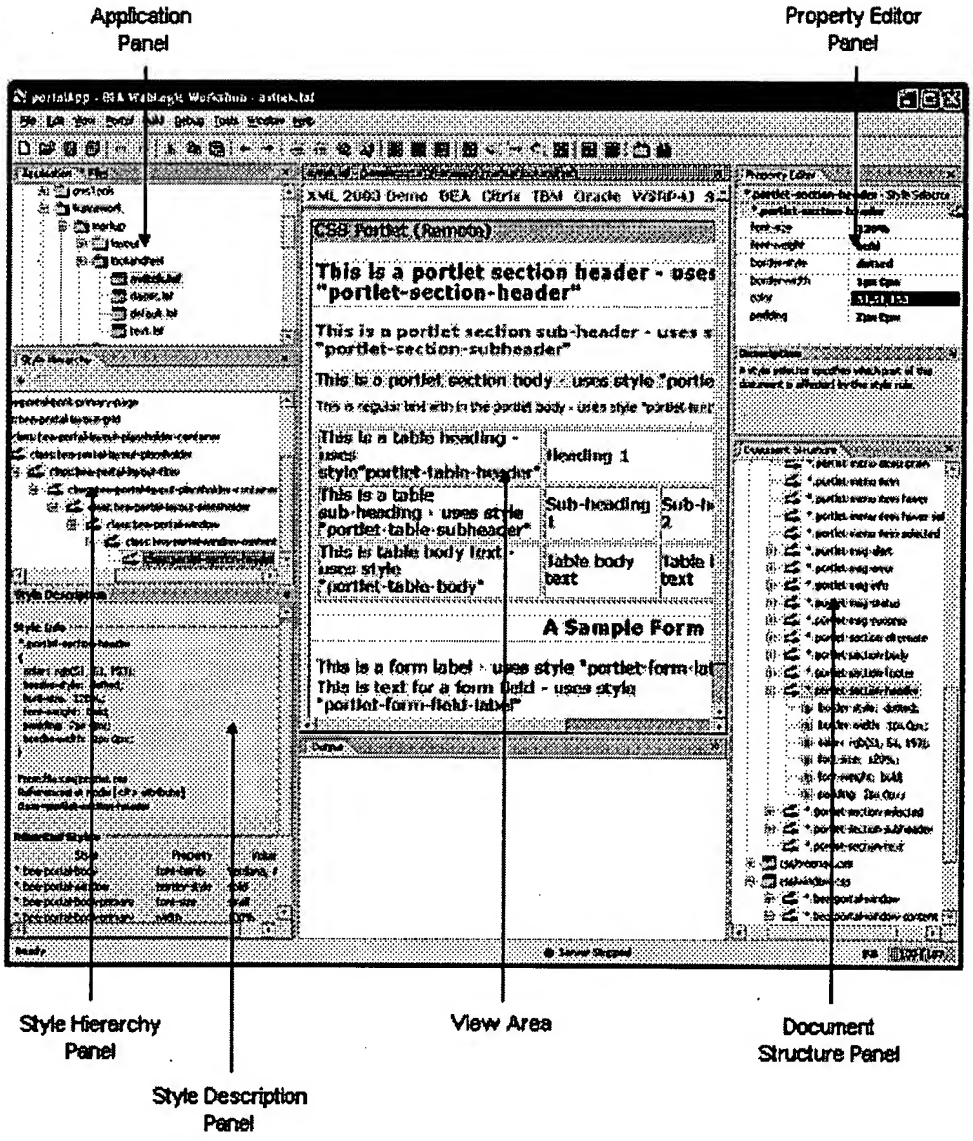
Overview

With the Look & Feel Editor, you can easily experiment with a portal's look & feel and see the results immediately. The Look & Feel Editor lets you interactively edit the text styles used by a portal. Using the Look & Feel Editor, you can select text in a portal and modify the text's characteristics, such as font size, color, padding, and so on. The changes you make are immediately reflected in the Editor's View Area.

Remember that a portal's skin helps to define the overall look & feel of a portal. The portal's `skin.properties` file specifies one or more CSS files used by the skin. A portal's HTML text can reference these CSS files and use their style definitions. If you modify the font size for a particular text style, the Look & Feel Editor changes the style's definition inside a CSS file. The change is then immediately reflected in the HTML displayed in the Editor's View Area.

The following figure shows the parts of the Look & Feel Editor. This topic discusses each of these parts in detail.

Figure 16 Look & Feel Editor components

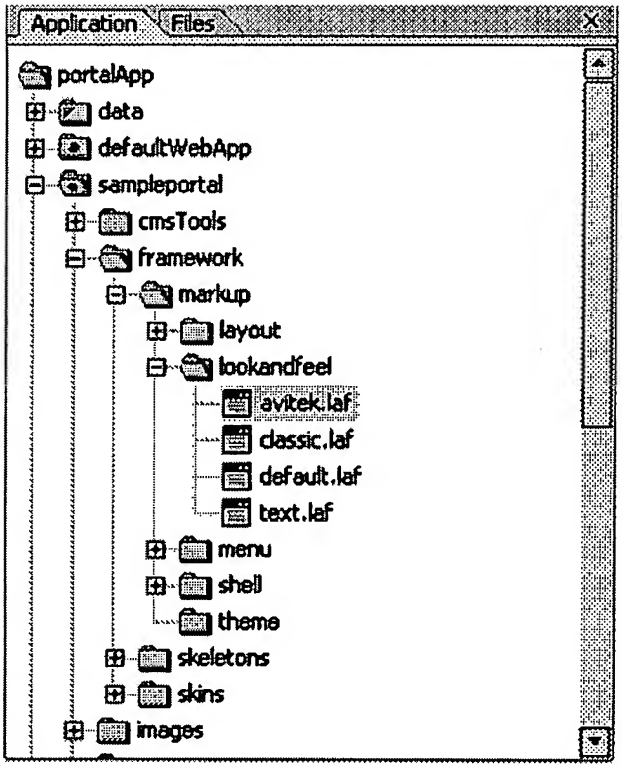


Application Window

The Application panel displays the file structure of a portal project. Use this panel to locate and select the look & feel file for the portal that you wish to edit.

The look & feel (`.laf`) file contains references to the skins and skeletons that define a portal's look & feel. To use the Look & Feel Editor, you must use the Application panel to locate the `.laf` file for the portal you wish to edit. Then, double-click the filename to open the Look & Feel Editor. The `.laf` files for a portal are located in the portal's `lookandfeel` folder. For example, the `avitek.laf` file is shown selected in the Application Window in the following figure:

Figure 17 Selected Look & Feel file

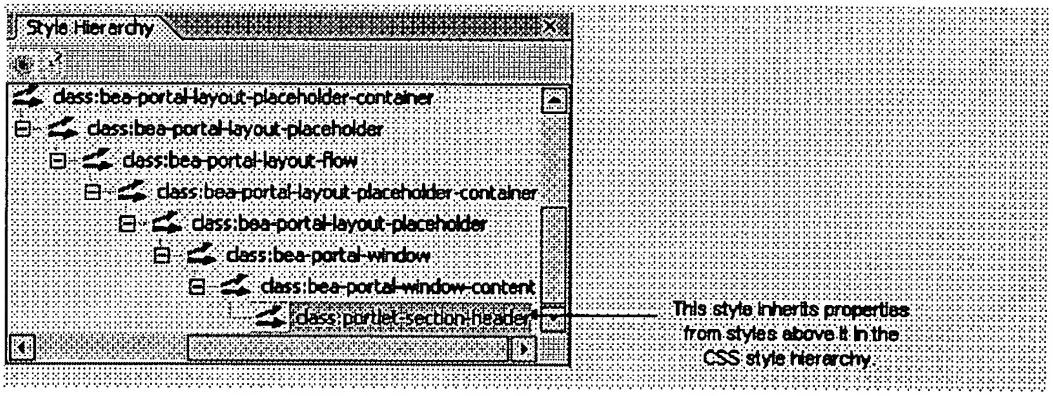


For more information on the `.laf` file, see The Look & Feel File.

Style Hierarchy Window

The Style Hierarchy panel shows the CSS cascade for the selected style. The cascade is a hierarchy of CSS styles, defined by the HTML document structure. It's useful to see the cascade because it can help you to locate and appropriately handle inherited style properties. In the following figure, the `portlet-section-header` style is selected. Note that the style `portlet-section-header` is below `bea-portal-window-content` in the hierarchy:

Figure 18 Selected CSS Style



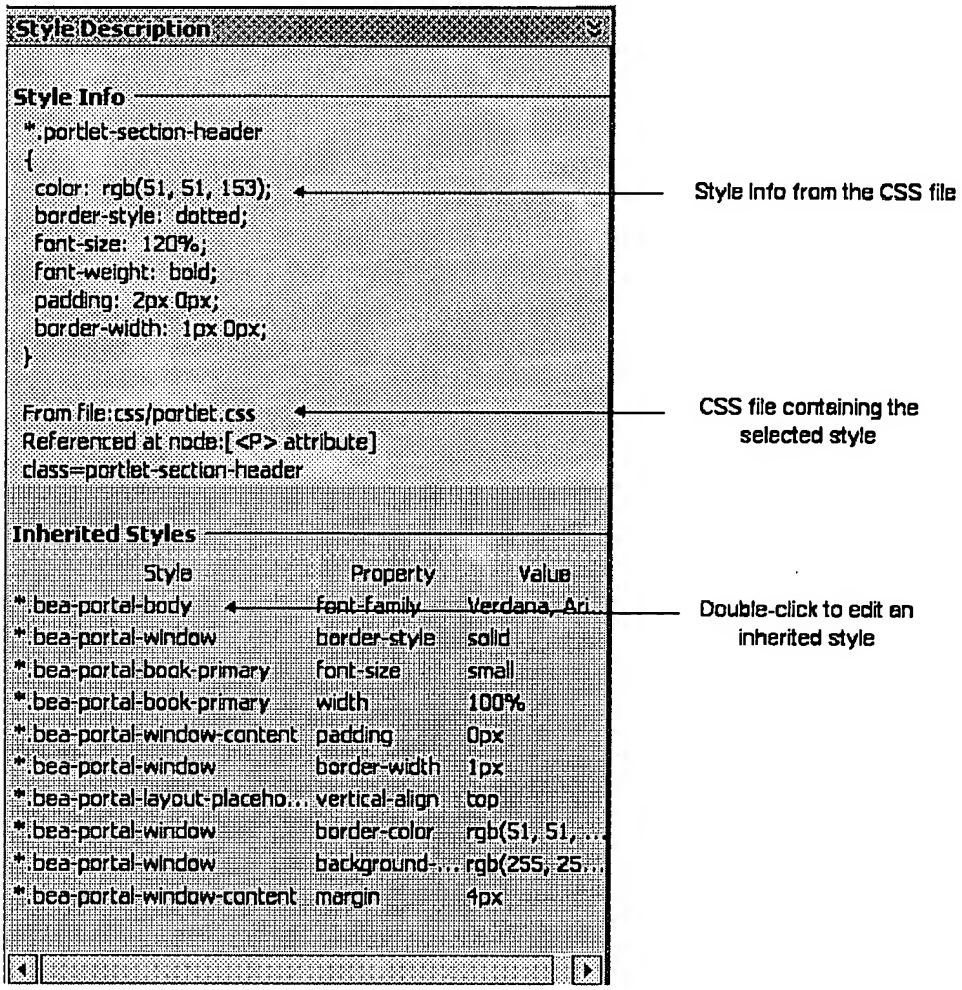
This means that `portlet-section-header` can inherit properties from `portal-window-`

content, and, potentially, from all other style classes higher up the hierarchy. For more information on inheritance, see Understanding CSS Inheritance. When you select a style in the Style Hierarchy panel, its style definitions and inherited style properties appear in the Style Description panel, described in the next section.

Style Description Window

The Style Description panel lets you see at a glance the selected style's properties and its inherited style properties. The **Style Info** part, shown in the following figure, comes directly from the CSS file in which the style is defined. The **Inherited Styles** list, also shown in the following figure, is constructed directly from the document structure of the HTML text that is currently opened in the Look & Feel Editor. The **Inherited Styles** list shows the style properties and their values that are inherited from styles higher up in the document hierarchy. For instance, you can see that `portlet-section-header` inherits the `font-family` property from the `bea-portal-body` style.

Figure 19 Window shows inherited styles



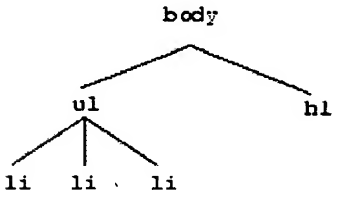
To understand the value of the **Inherited Styles** list, it helps to have a basic understanding of HTML and CSS.

Understanding CSS Inheritance

Tip: This section is a very brief overview of CSS inheritance. Many books and Websites are devoted to CSS and cover this important subject in greater depth.

HTML documents are hierarchically organized. In other words, each element of an HTML document can have one or more child elements, one parent element and possibly many ancestor elements. A central feature of CSS is that styles are *inherited* down the HTML document hierarchy. For example, the following tree diagram depicts a simple HTML document hierarchy:

Figure 20 CSS Inheritance



If you would like all the text in this document to be blue, you could define the `body` tag to be blue. Because of CSS inheritance, all of the elements below `body` (specifically, `li` and `h1`) will also be blue. If, on the other hand, you would like everything to be blue except list elements, you could define the `ul` tag to be another color, such as red. Then, all of the `li` elements will inherit the color red from their parent, `ul`. At the same time, the `h1` tags will be blue (`h1` tags will still inherit their color from `body`).

The Look & Feel Editor shows you all styles that a selected style inherits. Therefore, if you want to change the font size of a style, but font size is not defined in that style, you can see at a glance from which style font size is inherited. Then, you can easily edit the property, as explained in the next section.

Tip: Without this convenient feature, it would be difficult to decide which styles a given style inherited. Typically, you would have to open and examine the CSS files in the hierarchy to find where a specific style property is defined or possibly overridden.

Using the Inherited Styles List

As mentioned in the previous section, in some cases, the property you wish to modify is not defined in the specific CSS style class associated with the text you have selected. It is possible, for instance, to select a heading in the Look & Feel Editor, but find that font size is not a property of that heading's style. In this case, the property you wish to change may be an inherited property.

The Look & Feel Editor displays and lets you edit any inherited property for a given style. For example, suppose you wish to change the font size of some text. After selecting the style you wish to edit (for example, by clicking the text in the View Area), you then notice that `font-size` is not a property of that text's CSS style. Next you look at the **Inherited Styles** list, and you discover a style higher up in the cascade in which `font-size` is defined.

At this point, you must decide whether you want to edit the `font-size` property where it is currently defined (higher up in the cascade) or add the property directly to the style of the text you wish to modify. Of course, if you modify a property up the cascade, you may

inadvertently change the properties of other text that inherits the same property. It is up to you to make this decision. If you change it directly in the selected style, then the inherited property is overridden, and only that style (and any styles down the hierarchy) receive the new property value (unless it is once again overridden).

Tip: To add or modify a property in an inherited style, double-click the style name in the **Inherited Styles** list. Then, use the CSS Style Wizard to make your changes.

View Area

The View Area displays the HTML that uses the CSS styles you wish to edit. When you start the Look & Feel Editor, a default HTML page is displayed. This page is supplied with WebLogic Workshop and contains a representative sample of text elements.

You can load any other HTML into the Editor by supplying a URL or path, or by entering HTML directly. To select the HTML to display, use the Portal > Look And Feel menu. For example, to edit the text styles for a portal, run the portal in a browser, copy its URL, and load the page into the Editor using the menu function Portal > Look And Feel > Render Custom URL.

Note: Remember that you start the Look & Feel Editor by opening a look & feel (`.laf`) file. The HTML file that is shown in the View Area must reference the same CSS files that the `.laf` file references in its skin. If you load the default HTML page into the Editor, this connection is automatically established. However, if you load HTML from a portal into the Editor, you must be sure the portal references the same `.laf` file as the Editor.

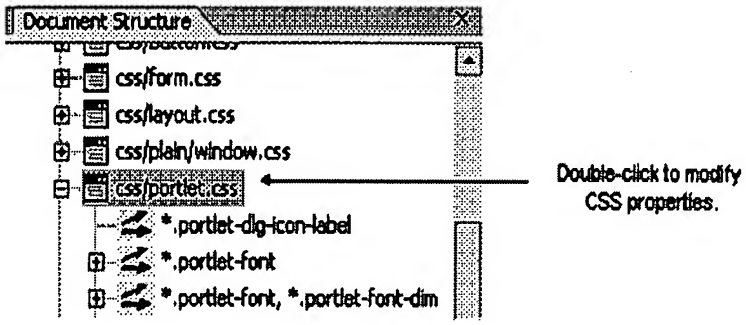
Document Structure Window

The Document Structure Window shows a representation of the files that are referenced by the portal's `skin.properties` file. In this panel you can edit properties of:

- The look & feel (`.laf`) file for the portal
- The style properties located in each of the CSS (`.css`) files referenced by the portal's skin

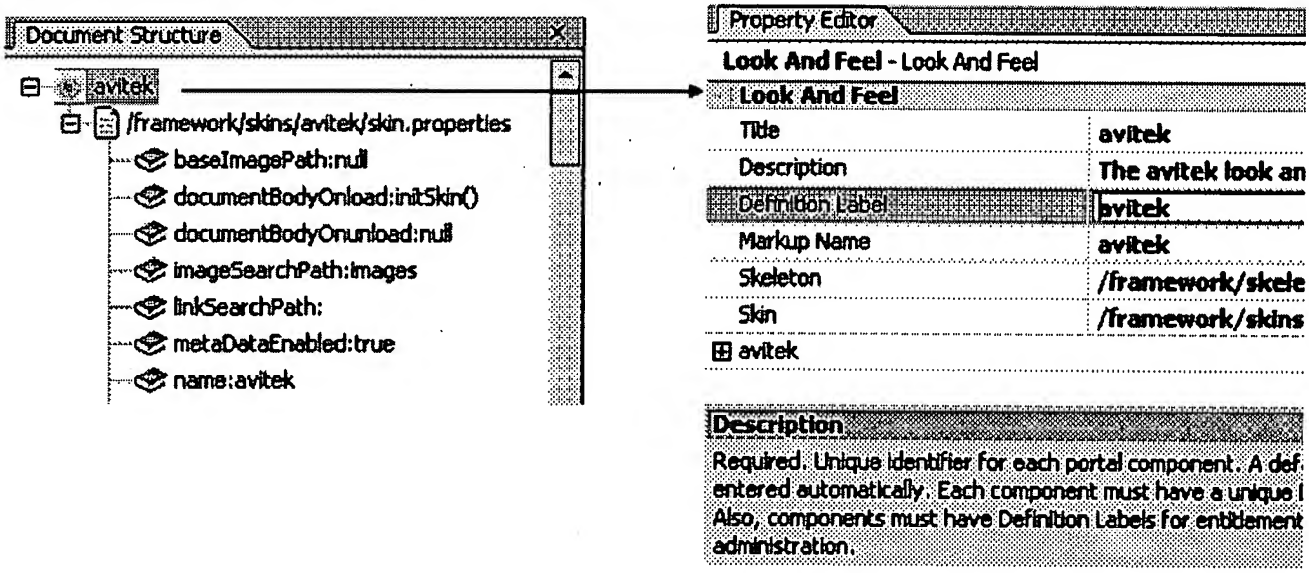
The following figure shows a portion of the Document Structure panel. In this figure, the `css/portlet.css` file is expanded to reveal the styles defined in it. You can double-click a style to add or modify its properties. You can also single-click a `.css` file, style name, or style property to display and edit values in the Property Editor panel.

Figure 21 Double-click a style to modify its properties



In addition to using this panel to access CSS styles, you can also access and edit the properties of the look & feel (`.laf`) file associated with a portal, as shown in the following figure. You can change any of these properties, including picking new skin and skeleton files. Note that the look & feel file node (e.g., avitek) occurs at the top of the document structure.

Figure 22 Look & Feel file in the Document Structure window



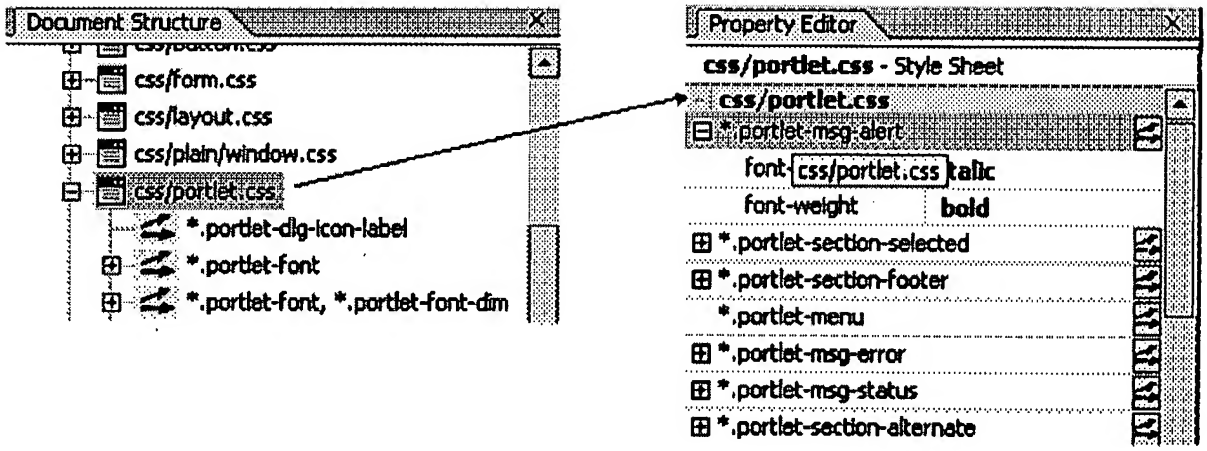
Note: If you select a new skin file, Weblogic Workshop automatically saves the currently open document. As a result, you will lose the ability to undo your most recent operations.

Property Editor Window

The Property Editor panel lets you interactively modify values of the selected CSS style or look & feel file. To display properties in the Property Editor, you can do one of the following:

- Click on a text element in the HTML file in the View Area.
- Click a CSS style or the look & feel filename in the Document Structure panel.
- Click on a CSS filename in the Document Structure panel, then expand the CSS file in the Property Editor to edit the properties, as shown in the following figure:

Figure 23 Displaying style properties in the Property Editor



Note: To see and edit values, you may have to expand the selected `.css` file or style by clicking a + icon on the left side of the Property Editor. To add or modify properties, you can also click the icon to the right of a style to bring up the CSS Style Wizard.

Summary

The Look & Feel Editor provides a convenient way to locate and edit the CSS-based styles that define the look of text elements in a portal. In addition, the Editor lets you modify the properties of the look & feel file associated with a portal.

back to top previous next

White Paper: WebLogic Portal Framework

Abstract

This document is intended to give an in depth overview of the technical underpinnings of the portal framework. It is targeted towards developers who have a deep J2EE background and are already familiar with WebLogic Portal.

This document is intended to be used as a supplement to the online documentation and assumes you have already immersed your self in it and are looking for more technical substance. This document will only discuss portal framework and not directly talk about the interaction with WebLogic Workshop, the WebLogic Administration Portal or any of the other features that come with WebLogic Portal; namely: Personalization, Campaigns, Content Management, Commerce Components, Entitlements and User Management. Many features described in this document have not made their way into WebLogic Workshop or the WebLogic Administration Portal as of this writing. Nonetheless, we encourage you to explore these features.

Definition of Terms, Acronyms, and Abbreviations

It is important that we first define a set of terminology, as we will be using these terms throughout the document. Please take the time to become familiar with these concepts.

Netui

Also called Page Flows, a programming model for building model 2 type applications. Netui is built on top of the popular Struts framework.

Netuix

An XML framework for rendering applications. Netuix was originally contrived as an extension to Netui. However, today netuix is no longer based on Netui nor dependant

on it. They are completely different technologies. Only the names are similar. With that said, netuix can seamlessly host netui applications.

Customization

The term used to modify a portal through an API. This API is typically called from our WebLogic Administration Portal and Visitor Tools pages but is also available to developers who whish to modify the desktop.

The API provides all the CRUD operations needed to modify a desktop and all of its components (Portlets, Books, Pages, Menus, and so on). Customization is different than Personalization. With Customization, someone is making a conscious decision to change the makeup of the desktop. With Personalization, changes are made based on rules and behavior (display an ad for Broncos tickets because it's Friday and the visitor lives in Denver).

Portal Framework

The portion of Weblogic Portal that is responsible for the rendering and Customization of the portal-what this document is all about.

Light Portal (File-based Portal)

A stripped down version of WebLogic Portal that does not deploy any EJBs or database. The light portal supports all the functionality in the portal framework with the exception of Customization. Light Portal can only render portal files, it cannot go to the database for a Customized desktop. Light portal rendering occurs in WebLogic Workshop in the development environment, but it may also be used in production systems.

UIControl

A netuix user interface control, not to be confused with business controls in WebLogic Workshop. Each element in the XML document (`.portal`, `.portal`, `.shell`, `.layout`, `.laf` and `.menu`) represents an instance ofa UI control. Typical controls are Books, Pages, Menus, Portlets, and so on.

Single File vs. Streamed Rendering

The .portal file you create in WebLogic Workshop is a fully functioning portal, however, it can also be used as a template to create a desktop. In this template you create books, pages and references to portlets and define defaults for them.

When you view the `.portal` file with your browser the portal is rendered in "single file mode," meaning that you are viewing the portal from your file system as opposed to a database. The `.portal` file's XML is parsed and the rendered portal is returned to the browser. The creation and use of a `.portal` is intended for development purposes and for static portals (portals that are not customized by the end user or administrator). Because there is no database involved you cannot take advantage of things such as user customization or entitlements.

Note: Externally, entitlements still run, they are just difficult to set.

Once you have created a `.portal` file, you can use it to create a desktop (streamed portal). A desktop is a particular view of a portal that visitors access. Desktops can be updated by administrators and end users.

A portal can be made up of multiple desktops, making the portal a container for desktops. A desktop contains all the portlets, content, shells, layouts, and look and feel elements necessary to create individual user views of a portal. When you create

a desktop based on the .portal file in the WebLogic Administration Portal, a desktop and it's books and pages are placed into the database.

The desktop, books and pages reference shells, menus, look and feels and portlets. The settings in the .portal file, such as the look & feel, serve as defaults to the desktop. Once a new desktop is created from a .portal template, the desktop is decoupled from the template, and modifications to the .portal file do not affect the desktop, and vice versa.

For example, when you change a desktop's look & feel in the WebLogic Administration Portal, the change is made only to the desktop, not to the original .portal file. When you view a desktop with a browser it is rendered in "streaming mode" (from the database). Now that a database is involved, desktop customizations can be saved and delegated administration and entitlements can be set on portal resources.

Library

The library is a home for a set of public controls that are not associated with a desktop. In other words Books, Pages, Portlets, can be created and modified outside the scope of a desktop and then later added to a desktop. Changes to objects in the library are cascade down through the desktops and user customizations.

Netuix

Controls

As stated earlier, netuix is an XML framework for rendering applications, whether these applications look like portals or not. Many customers who use our product today create applications from our framework that look nothing like a portal. Typically when people think of portals they think of "My Yahoo!". While many applications developed with netuix look like My Yahoo!, many do not.

A netuix application is represented by one or more XML documents, the most familiar being the .portal file (an XML file with a .portal extension). This portal file may or may not include other portal include files, called .pinc files for short (files with the extension .pinc). Just like a JSP can include other JSP files to distribute functionality, a portal file can include other portal files.

A .pinc file is different from a portal file in that a portal includes the root elements or controls while the .pinc file does not. We will discuss this in more detail later. However, for this discussion the portal file is the parent, and it may in turn include one or more .pinc files, which in turn may include other .pinc files. One other important note: a .pinc file must begin with a Book or a Page element as the root element. More on what Books and Pages are in a bit.

In the portal file, you can think of each element representing an instance of a UI control. (UIControls are not to be confused with business controls in Workshop.)These controls are wired in a hierarchical tree. In other words, each control has a parent and zero or more children. The controls can discover each other at runtime and can modify the tree by adding new children or removing existing children. All controls run through a lifecycle (a set of methods called on the control in a particular order). All the methods are called in turn in a

depth first order.

To best illustrate this, let's walk through the sequence of events that happen when a person requests a portal in single file mode from the browser. But before we do that, we first need to cover a few architectural issues with the portal framework. All requests for a portal or desktop come in through the PortalServlet. The PortalServlet is registered in the web.xml file under the url-patterns appmanager and *.portal. If the PortalServlet detects a request ending with `.portal` it knows the request is for a locale file and does not need to go to the persistence API for the XML.

The first thing the PortalServlet must do is parse the XML file (`.portal`) and generate a control tree from it. Every element in the portal file represents a control in the control tree, and every attribute on the element represents an instance variable on the control. The same hierarchy is maintained in the XML document as in the control tree. A control is simply a Java class that extends another Java class, namely the UIControl class. In this release we don't explicitly expose controls to developers, but developers can interact with the controls using backing files, context, and skeleton JSPs. This is discussed later.

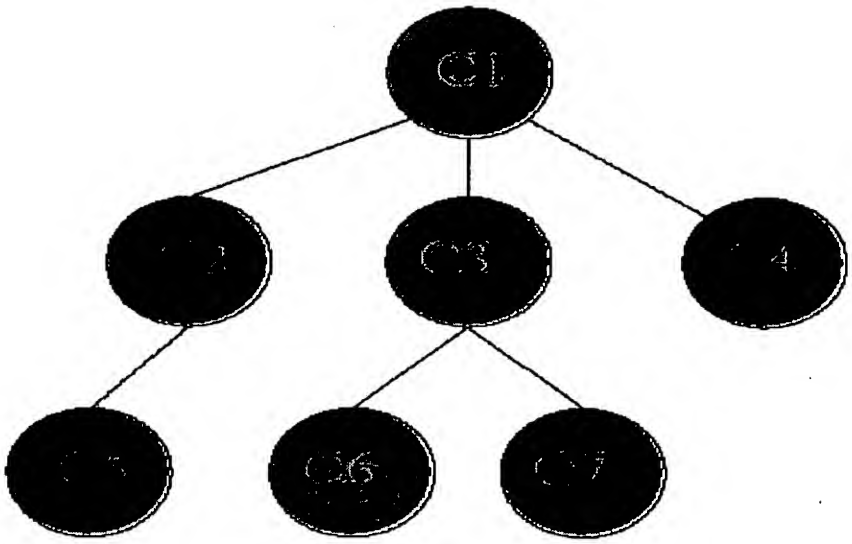
Note: The PortalServlet doesn't actually parse the XML document on each request. A lot of caching and magic is going on behind the scenes to get the desired performance for the enterprise applications.

Once the control tree is built and all the instance variables are set on the controls, the control tree is run through its lifecycle. The lifecycle can be thought of as a set of methods on the controls that are called on in a well-defined order. The lifecycle methods are as follows:

```
init()
loadState()
handlePostbackData()
raiseChangeEvents()
preRender()
saveState()
render()
dispose()
```

These methods are called in depth first order. In other words, all the `init()` methods are called, followed by the `loadState()` methods, and so on. They will also be called depth first. Example, given the following control tree, the order in which the `init()` method would be called is: C1, C2, C5, C3, C6, C7, C4, then the `loadState()` method would be called in the same order, and so on.

The last method to be called would be C4's `dispose()`:



Portal Controls

This section describes all the netuix controls that make up the portal framework. The control relationship is driven by the XML schema definition `controls-netuix-1_0_0.xsd`. The following figure summarizes the schema definition.

Desktop

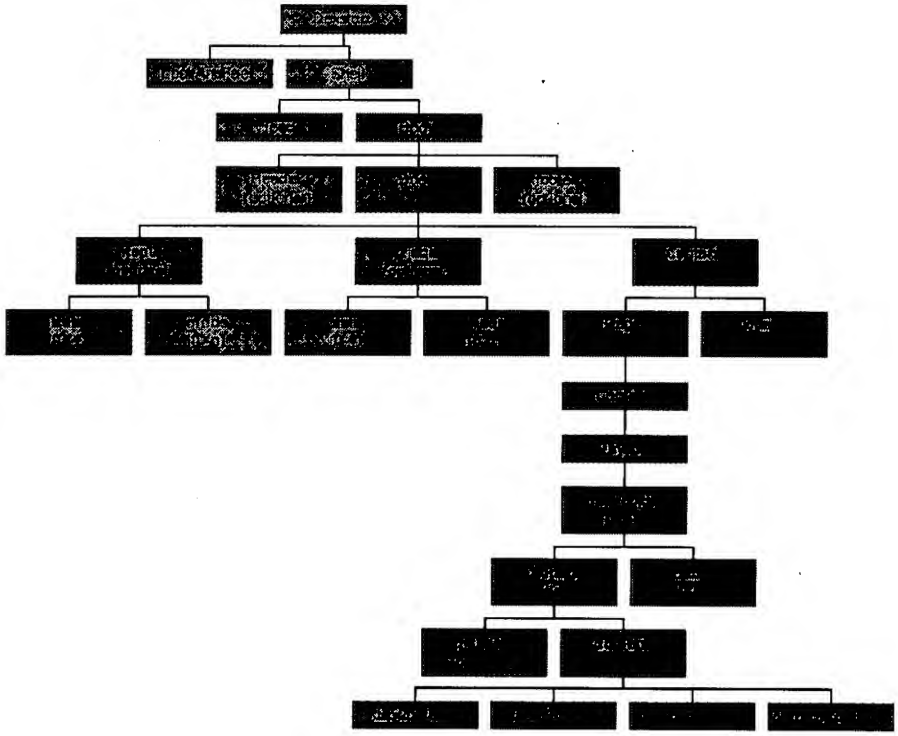
The Desktop control is the parent control that hosts all the other netuix controls. Every portal must have one Desktop control. The Desktop control actually provides little functionality above and beyond entitlement checking and a place to go to discover other controls.

The most important use of the Desktop control from a developer perspective is that it has a `PresentationContext` that can be traversed to get references to all the child controls, like books, pages, and portlets. A `DesktopBackingContext` was added in 8.1 sp3 along with a richer set of methods for locuting child controls.

Windows

A Window control provides functionality similar to the windowing concept on your computer. Windows support States and Modes. States affect the rendering of the Window, like minimize, maximize, float, and delete. Modes affect the content, like Edit and Help. (Custom modes are also supported.) Windows can also act as a container for other Windows. For example, a book can contain a page.

All Window controls must have a Content control. The Content control is responsible for hosting the actual content inside the window. The Window control is an abstract class that is one of the three derived classes that must be used in the portal. These derived classes are: Books, Pages and Portlets. The figure below shows the relationship between Windows, Books, Pages and Portlets.



Book

A Book aggregates a set of navigables. A navigable is a Book or a Page. A Book may have an optional menu control that provides navigation among navigables. From a code standpoint, Navigable is an interface that Book and Page implement.

Page

A Page is used to display a set of Placeables. A Placeable is a Portlet or Book. The Page has a layout which has one or more Placeholders which can host zero or more Placeables.

Portlet

Portlets are used as windows to host may different types of applications. As of this writing the applications can be any one of following: HTML pages, JSP files, `.pinc` files, Page Flows, Struts, Webflows, JSR 168 Portlets, and WSRP proxy portlets.

Menus

Menus are optional components that are loosely coupled to books and pages. A menu is responsible for displaying some type of navigation component, whether it is a set of tabs, a set of links, or some tree structure. The menu fires `PageChaneEvents` that the Pages themselves listen to and activate accordingly.

At the time of this writing, WebLogic Portal provides two types of menus: singlelevel and multilevel. Future service packs and releases may include more. You can also create you own menus by using JSPs and the `<render:pageUrl/>` tag, or from a backing file call the

`setupPageChangeEvent` method an a Book, Page or Portlet backing context before the `preRender` method.

SingleLevelMenu

Provides a single row of tabs for the book's immediate pages and child books.

MultiLevelMenu

Recursively provides a hierarchical menu for all the books and pages contained within a book. This menu does not stop at the first set of children. It continues down the tree. If the parent book uses a `multilevelmenu`, then the child books should not use a menu as the `multilevelmenu` will cover them.

Layouts

Layouts and Placeholders (not to be confused with personalization placeholders) are used to structure the way portlets and books are displayed on a page. Layout placeholders are rendered as HTML table cells.

WebLogic Portal ships with some predefined layouts and the ability to create your own custom layouts. More layouts will probably be shipped in future service packs and future releases. If the supplied layouts don't meet your needs, you will have to create your own custom layout. The next section describes that process in detail.

Information for Creating a Custom Layout

When creating a custom layout you will need to create three things:

- The Layout File
- The html.txt File
- The Skeleton JSP

The Layout File

The layout file contains the snippet of XML that describes the controls that make up the layout. The markup from this file is what gets copied into the `.portal` file and into the database for reassembly. A layout file must have a `.layout` extension and can live anywhere in the Web application directory except `WEB-INF`.

Note: Changes made to a layout file after it has been created get picked up automatically in the database but will not automatically update the layout in the `.portal` files. This is because the `.portal` file contains a copy of the markup and not a reference to it.

The `.layout` files must be created by hand (text or XML editor). The best way to get started is by copying an existing layout. Layout files shipped with the portal are located in the /framework/markup/layout directory.

Layout File Elements

The parent element for all markup types is:

```
<netuix:markupDefinition/>
```

This parent element has two child elements:

- `<netuix:locale language="nn" [country="nn"] [variant="nnnn"]/>`

  Defines the working local for the title and description attributes defined later.

- `<netuix:markup>`

  Defines the outer envelope stanza that marks the beginning and end of the XML that will define this layout.

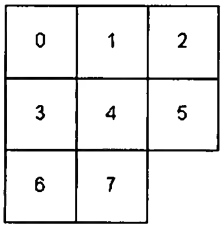
Basic Layout Controls

The next set of elements are unique to a layout. When creating your own layout you will have to choose from one of these four base layout controls. The `<netuix:layout/>` is the most generic of the four and all others are derived from it. The `<netuix:layout/>` control provides the most flexibility but is also the most difficult to implement.

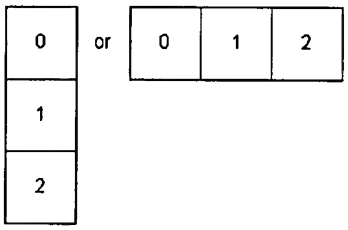
- <netuix:layout title="" [description=""] type = "" htmlLayoutUri="" [iconUri=""] markupName="" markupType="Layout" [skeletonUri=""] [properties=";"]/>
- <netuix:gridLayout columns="(1-n)" [rows = "1-n"] /> Attributes
- <netuix:borderLayout [layoutStrategy="order | title"] /> Attributes
- <netuix:flowLayout [orientation="vertical | horizontal"]/> Attributes
- <netuix:placeholder title="" [description=""] [flow="horizontal | vertical"] [usingFlow=""] [width=""] markupName="" markupType="Placeholder" [skeletonUri=""]/>

<netuix:layout title="" [description=""] type = "" htmlLayoutUri="" [iconUri=""] markupName="" markupType="Layout" [skeletonUri=""] [properties=";"]/>

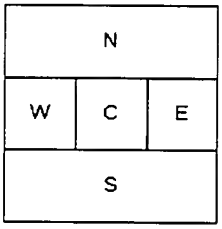
This is the base control for all layouts. This control can be used directly or you can use one of the following three derived controls.



Grid Layout
The grid layout automatically positions the number of placeholders you specify into the number of columns and rows you specify. This examples sets columns="3" to position 8 Placeholders



Flow Layout
The flow layout automatically positions the number of placeholders used either vertically or horizontally with no wrapping.



Border Layout
The border layout lets you use up to five placeholders. You can position the placeholders with the attributes "north," "south," "east," "west," and "center."

Attribute	Description
title	This is the internationalized title displayed to the user and administrators when selecting the layout they want to use. **Note:** The developer only works in one language as defined in the `<netuix:locale>` element described previously. More internationalized versions of the title and description can be added later with the WebLogic Administration Portal.
description	An optional internationalized description of your layout.
type	The type of layout. This is hard coded for the three derived layouts. If you create a custom come up with your own type.
htmlLayoutUri	A fully qualified path (from the top of the Web application) to the html.txt file to be used by the WebLogic Administration Portal.
properties	"name/value pairs that can be passed to the skeleton as hints. These properties can be separated using a semicolon ";".
iconUrl	A fully qualified path (from the top of the Web application) to the `.gif` file to be used by the WebLogic Administration Portal.
markType	This field is required and must be "Layout".
markupName	This field is required and must be unique per Web application. If you copied the XML from another layout you must change this name.
skeletonUri	A fully qualified path (from the top of the Web application) to the skeleton JSP to be used for runtime rendering.
presentationClass	Optionally provides a generic presentation "class," such as a CSS class, for use by external rendering devices.
presentationStyle	Optionally provides a generic presentation "style," such as a CSS style, for use by external rendering devices.
presentationId	Optionally provides a generic presentation "id" for use by external rendering devices.

<netuix:gridLayout columns="(1-n)" [rows = "1-n"] /> Attributes

This layout defines a grid where you can specify the number of columns and rows. This layout is typically used to create one, two , three, column layouts.

Attribute	Description
columns	A required attribute that identifies the number of columns in the grid
rows	An optional attribute specifying the number of rows in the grid. If this attribute is omitted then the default one row will be used

<netuix:borderLayout [layoutStrategy="order | title"] /> Attributes

This layout has four border placeholders and one center placeholder. The north and south placeholders span the length of the table. The west, center, and east placeholders comprise the middle row and have respective widths of 25, 50, and 25 percent. The north and south portlets flow horizontally in the placeholders, and the others flow vertically.

Attribute	Description
layoutStragety	Defines what placeholder will be the north, west, center, east and south. If "title" is specified then each placeholders must specify the correct title.

<netuix:flowLayout [orientation="vertical | horizontal"]/> Attributes

A layout that just flows the contents in a vertical or horizontal fashion.

Attribute	Description
orientation	Flow the contents vertical or horizontal. Layout controls contain one or more child placeholder controls. These controls have the following attributes.

<netuix:placeholder title="" [description=""] [flow="horizontal | vertical"] [usingFlow=""] [width=""] markupName="" markupType="Placeholder" [skeletonUri=""]/>

Placeholders are child elements of the above four types of layouts. Every layout must have at least one placeholder child element. Each placeholder has a "layout location." Thie loayout location is defined by its position inside the layout files. Layout locations start at 0, 1, 2.

Attribute	Description
title	This is the internationalized title displayed to the user and administrators when selecting the placeholder they want to use. **Note:** The developer only works in one language as defined in the <netuix:locale> element described above. More internationalized versions of the title and description can be added later using the WebLogic Administration Portal.
description	An optional internationalized description of your placeholder.
markType	This field is required and must be "Placeholder".
markupName	This field is required and must be unique per Web application. If you copied the XML from another layout you must change this name. The naming convention is layoutMarkupName_placeholdersName, but you can use what

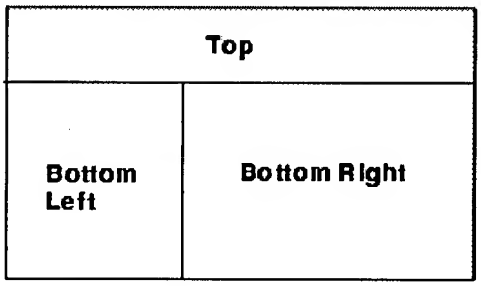
	you want as long as it is unique.
skeletonUri	A fully qualified path (from the top of the Web application) to the skeleton JSP to be used for runtime rendering. Typically the default skeleton will suffice for all custom layouts, but you have the option to create your own.
flow	An optional value specifying the direction of content flow; default is "vertical."
usingFlow	An optional value specifying whether or not flow should be used; default is "true."
width	An optional hint attribute to tell the parent layout how much width this placeholder wishes to have allocated.
properties	Name/value pairs that can be passed to the skeleton as hints. These properties can be separated using a semicolon ";". Example: `properties="rowspan=2;columnspan=3;myprop=hello"`

Example of a Custom Layout

Now that we have described the four basic layout controls and child placeholder controls, you will have to choose which one of these to base your custom layout on. Unless you are tweaking one of the parameters in the three sub-class layout controls, you may want to choose the `<netuix:layout>` control.

The easiest way to describe how to create a custom layout is to give an example. Lets create a custom layout with a spanning row at the top with two columns underneath. The two columns will split the real estate in a 30%-70% fashion.

Our layout will look something like this:



1. The first thing we need to do is create a layout file (again the easiest way is to copy one from another layout).

   We will call our layout file `spanningtwocolumn.layout`, and it will look something like this:

Listing 1 Sample Code for a Layout File

```
<netuix:layout title="Spanning Two Column" description="One row and two columns.
    type="spanning"
    skeletonUri="/customskeletons/spanningtwocolumnlayout.jsp"
    htmlLayoutUri="/framework/markup/layout/spanningtwocolumn.html.txt"
    iconUri="/framework/markup/layout/spanningtwocolumn.gif"
    markupType="Layout" markupName="spanningTwoColumnLayout">
        <netuix:placeholder title="top" description="The top spanning placeholde
                            markupType="Placeholder"
                            markupName="spanningTwoColumn_top">
        </netuix:placeholder>
        <netuix:placeholder title="left" description="The bottom left placeholde
                            markupType="Placeholder"
                            markupName="spanningTwoColumn_left">
        </netuix:placeholder>
        <netuix:placeholder title="right" description="The bottom right placehol
                            markupType="Placeholder"
                            markupName="spanningTwoColumn_right">
        </netuix:placeholder>
</netuix:layout>
```

The <netuix:markupDefinition>, <netuix:locale/> and <netuix:markup/> elements were left out of the example for the sake of clarity. DON'T forget to include these in your .layout file.

In the previous layout example we are using the <netuix:layout/> element, and we have three placeholders underneath it. Other things to note are the markupNames are unique, and we have identified our own custom skeleton to do the rendering.

The Skeleton JSP

Since a custom skeleton is being used to do the rendering (as specified by the skeletonUri attribute) this JSP needs to be created. Again, the easiest way is to copy an existing one.

Note: The skeleton JSP for the control is called twice: once during "begin render" and once for "end render." Between the begin render and end render phase the children are rendered. This allows you to start HTML tables in the begin render section and close them in the end render section. All skeleton files should have the following JSP tags:

- <render:beginRender></render:beginRender>
- <render:endRender></render:endRender>.

The body of the beginRender tag is only evaluated during the begin render phase, and the body of the endRender tag is only evaluated during the end render phase.

Here is what our new skeleton JSP (/customskeletons/spanningtwocolumnlayout.jsp) will look like

Listing 2 New Skeleton JSP

```
%@ page import="com.bea.netuix.servlets.util.RenderToolkit,
com.bea.netuix.servlets.controls.layout.LayoutPresentationContext,
java.util.List,
                com.bea.netuix.servlets.controls.layout.PlaceholderPresentationC
%>
<%@ taglib uri="render.tld" prefix="render" %>
<%
    RenderToolkit toolkit = RenderToolkit.htmlInstance();
```

```
    LayoutPresentationContext layout =
        LayoutPresentationContext.getLayoutPresentationContext(request);
%>
<render:beginRender>
    <table
        <% toolkit.writeId(out, layout.getPresentationId()); %>
        <% toolkit.writeAttribute(out, "class", layout.getPresentationClass(), "
        cellspacing="0"
    >
        <tbody>
    <%
        List children = layout.getChildren("layout:placeholder");
        // Could get optional properties here to help with rendering
        // String property = layout.getProperty("myProperty");
        for (int i = 0; i < children.size(); i++)
        {
            PlaceholderPresentationContext placeholderPresentationContext =
                (PlaceholderPresentationContext)children.get(i);
            if (i == 0)
            {
     %>
                <tr>
                <td colspan="2" width="100%" valign="top" class="layout-placeho
                    <% toolkit.renderChild(placeholderPresentationContext, reque
                </td>
                </tr>
     <%
            }
            else if (i == 1)
            {
     %>
                <tr>
                <td width="30%" valign="top" class="layout-placeholder-containe
                    <% toolkit.renderChild(placeholderPresentationContext, reque
                </td>
    <%
            }
            else if (i == 2)
            {
    %>
                <td width="70%" valign="top" class="layout-placeholder-containe
                    <% toolkit.renderChild(placeholderPresentationContext, reque
                </td>
                </tr>
    <%
            }
        }
    %>
</render:beginRender>
<render:endRender>
        </tbody>
    </table>
</render:endRender>
```

Note: This example the widths are hard coded in the JSP. Instead, these widths should be specified in the layout file as an attribute to the placeholder. The widths can then be referenced in the skeleton as follows:

```
<render:writeAttribute name="width" value="<%=
```

```
placeholderPresentationContext != null ?
placeholderpresentationContext.getWidth() : null %>"/>
```

Also, other properties like "rowspan=2" can be passed as name/value pairs on the properties attribute and "endtop" to create a more generic row/column spanning layout.

The custom layout is now functionally complete. The html.txt file has not yet been created, but the layout can be tested. To do this, start WebLogic Workshop, open or create a portal file, select a page, and in the Property Editor window select the custom layout in the Layout field.

Note: if you change your `.layout` file after you have used it in the `.portal` file, changes won't be reflected in the .portal file. This happens because when you use a layout in the .portal it copies the markup from the layout. You will need to chooses another layout and then choose the original one back again to see the changes.

The html.txt File

The `.html.txt` is an `HTML` snippet strictly used by the WebLogic Administration Portal and Weblogic Workshop to give a visual representation of what the layout looks like, so the administrator can place the portlets in the correct placeholders. Typically this is the last file you will create, because it is not used by the rendering framework.

The last thing to do is create the `html.txt` file so the WebLogic Administration Portal can provide a visual representation of the layout.
The `/framework/markup/layout/spanningtwocolumn.html.txt` should look something like this:

Listing 3 Sample html.txt Code

```
<table class="portalLayout" id="thePortalLayout" width="100%" height="100%">
  <tr>
    <td class="placeholderTD" valign="top" width="100%" colspan="2">
      <placeholder number="0"></placeholder>
    </td>
  </tr>
  <tr>
    <td class="placeholderTD" valign="top" width="30%">
      <placeholder number="1"></placeholder>
    </td>
    <td class="placeholderTD" valign="top" width="70%">
      <placeholder number="2"></placeholder>
    </td>
  </tr>
</table>
```

Interacting With UI Controls

Since controls are not exposed directly to developers, developers need a way to directly interact with and affect the behavior of the controls. To accomplish this, WebLogic Portal exposes context, backing files, skeletons, and events. Developers should use these components when trying to alter the behavior of or interact with the portal framework.

Context

A context is nothing more then a delegate to the underlying control. This delegate only exposes the supported methods on the control.

Contexts are broken down into two types: backing context and presentation context. Backing contexts are available from the backing files, and presentation contexts are available from the JSPs.

Two types of context are required because certain methods apply at certain times in the lifecycle. For example, it doesn't make sense to have a `setTitle()` method on the Presentation context because the portal has already started to render and it would have no effect. Calling this method from a backing file, however, is appropriate.

Backing Context

BackingContext is available from backing files. A reference to a Backing context can be obtained in one of two ways:

- The first way is to use the static method `getXXXBackingContext` on the context class. This method will return the active backing context for that type. To be more specific, if I call this method from portlet A's backing file, I will get the backing context for portlet A not portlet B.

  Similarly, if I call `getPageBackingContext(request)` from portlet A, I will get the page backing context for the page portlet A is located on.

- The second way to obtain a backing context is from another context. This can be useful when you want a context that is not the active context. Example would be, I want to obtain portlet Bs backing context from portlet A.

  If portlet A is contained within the same page as Portlet B then one could use:

  ```
  PortletBackingContext portletB = PageBackingContext.getPageBackingContext
  (request).PortletBackingContext getPortletBackingContextRecursive("Portlet
  Bs instance label");
  ```

  If Portlet A does not know where portlet B is located then you can delegate to the DesktopBackingContext

  ```
  PortletBackingContext portletB =
  DesktopBackingContext.getPageBackingContext(request).PortletBackingContext
  getPortletBackingContextRecursive("Portlet Bs instance label");
  ```

Refer to the javadoc on these and other backing context for more information.

`com.bea.netuix.servlets.controls.page.PageBackingContext`

`com.bea.netuix.servlets.controls.application.backing.DesktopBackingContext`

Presentation Context

PresentationContext are available from `JSP` files. A reference to a presentation context can be obtained in one of two ways:

- The first way is to use the static method getXXXPresentationContext on the context class. This method will return the active presentation context for that type. To be more specific, if I call this method from portlet A's content JSP, I will get the presentation context for

portlet A not portlet B. Similarly, if I call `getPagePresentationContext(request)` from portlet A, I will get the page Presentation context for the page portlet A is located on.

- The second way to obtain a presentation context is from another context. This can be useful when you want a context that is not the active context. For example, I want to obtain portlet Bs presentation context from portlet A.

Backing Files

Backing files are simple Java classes that implement the `com.bea.netuix.servlets.controls.content.backing.JspBacking` interface or extend the `com.bea.netuix.servlets.controls.content.backing.AbstractJspBacking` abstract class (in retrospect it should have been called a backing class). The methods on the interface mimic the controls lifecycle methods and are invoked at the same time the controls lifecycle methods are invoked.

The controls as of this writing that support backing files are:

- Books
- Pages
- Portlets
- JspContent controls.

Note: Desktops also support backing files as of Service Pack 3

A new instance of a backing file is created per request, so you don't have to worry about thread safety issues. New Java VMs are specially tuned for short-lived objects, and this is not the performance issues it once was in the past. Also JspContent controls support a special type of backing file that allows you to specify if the backing file is thread safe. If this value is set to true, only one instance of the backing file is created and shared across all requests.

Skeletons

Skeletons are JSPs that are used during the render phase. The render phase is actually broken into two parts: begin render and end render. The parent control's begin render is called, followed by its children's begin render, their children's begin render, and so on. After the last begin render is called, the children's end renders are called, ending with the parent's end render. This allows the parent to provide a container, such as an HTML table, and the children to provide the table contents.

Each skeleton is actually called twice. There are special tags in the skeleton that only evaluate to true depending on which render phase you are in.

Events

There are four types of events in the system:

- Window Mode
- Window State
- Page Change

- Generic Portlet Events.

The Mode, State and Page Change Events are not exposed directly to the developer but can be configured through special methods on the Window backing files. Namely: `setupModeChangeEvent`, `setupStateChangeEvent`, and `setupPageChangeEvent()`. The methods must be called before the `preRender` method as events are fired just after `handlePostbackData` method. They will also only work if the `handlePostbackData` method returns true (see javadoc).

Note: When calling one of the `setupxxevent` methods, it must be done on the backing context that the backing file is tied to. If you do no do this the event may not get fired.

Portlet Events (not to be confused with page flow events) allow portlets to communicate. One portlet can create an event and other portlets can listen for that event. These Portlet events can also carry payloads.

Here is an example of one portlet firing and event from a backing file and other portlets listening for the event:

Listing 4 Sample Code for Portlet Firing and Event from a Backing File

```
/**
 * This is the implementation on the backing file of the portlet that wants to f
 */
public boolean handlePostbackData(HttpServletRequest request, HttpServletRespons
{
   // Create a new portlet event with the results in the paylod
    PortletEvent portletEvent = new PortletEvent(new MyPayload("Hello From portl

   // Get a hold of the portlet event manager and fire the event.
   PortletBackingContext portletBackingContext =
        PortletBackingContext.getPortletBackingContext(request);
   PortletEvent.Manager portletEventManager =
        PortletEvent.getEventManager(this, portletBackingContext);
   portletEventManager.fireEvent(portletEvent);
   // Needed for the event to fire.
    return true;
}
/**
 * This is the implementation of the portlet that wants to receive the event.
 */
public class ResultBacking extends AbstractJspBacking implements PortletEventLis
{
    MyPayload result;
    public void init(HttpServletRequest request, HttpServletResponse response)
    {
        result = null;
        // Register for Portlet Events
        PortletBackingContext portletBackingContext =
            PortletBackingContext.getPortletBackingContext(request);
        PortletEvent.addGlobalListener(portletBackingContext, this);
        CustomPortletEvent.Manager portletEventManager =
            CustomPortletEvent.getEventManager(this, portletBackingContext);
    }
    public void handleEvent(Object source, AbstractEvent event)
    {
        // Can check the source of the event
        if (source instanceof PortletA)
```

```
            {
                result = (MyPayload)((PortletEvent)event).getPayload();

        }
    }
```

Note: The tutorial portal contains examples of portlets communicating via events.

Customization

Customization is the term used to describe Administrators and End Users making modifications to a desktop. Normally this is done through the Administration tools or the Visitor Tools web application. However, these API can be called directly from within the developer's code. For additional information on these API refer to the Javadoc.

back to top previous next